Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations

Matthieu Dorier[a,*], Gabriel Antoniu[b], Franck Cappello[c], Marc Snir[c,d], Roberto Sisneros[d], Orçun Yildiz[b], Shadi Ibrahim[b], Tom Peterka[c], Leigh Orf[e]

[a] *ENS Rennes, IRISA, Rennes, France*
[b] *Inria, Rennes - Bretagne Atlantique Research Centre, France*
[c] *Argonne National Laboratory, Lemont, IL, USA*
[d] *University of Illinois at Urbana Champaign, IL, USA*
[e] *Central Michigan University, MI, USA*

Abstract

With exascale computing on the horizon, reducing performance variability in data management tasks (storage, visualization, analysis, etc.) is becoming a key challenge in sustaining high performance. This variability significantly impacts the overall application performance at scale and its predictability over time.

In this paper, we present Damaris, a system that leverages *dedicated cores* in multicore nodes to offload data management tasks, including I/O, data compression, scheduling of data movements, in situ analysis and visualization. We evaluate Damaris with the CM1 atmospheric simulation and the Nek5000 computational fluid dynamic simulation on four platforms, including NICS's Kraken and NCSA's Blue Waters. Our results show in particular that (1) Damaris fully hides the I/O variability as well as all I/O-related costs, which makes simulation performance predictable; (2) it increases the sustained write throughput by a factor of up to 15 compared with standard I/O approaches; (3) it allows almost perfect scalability of the simulation up to over 9,000 cores, as opposed to state-of-the-art approaches that fail to scale; (4) it enables a seamless connection to the VisIt visualization software to perform in situ analysis and visualization in a way that does not impact the performance of the simulation, nor its variability.

In addition, we further extended our implementation of Damaris to also support the use of *dedicated nodes* and conducted a thorough comparison of the two approaches –dedicated cores and dedicated nodes– for I/O tasks with the aforementioned applications.

Keywords: Exascale Computing, I/O, In Situ Visualization, Dedicated Cores, Dedicated Nodes, Damaris

*Corresponding author
Email address: matthieu.dorier@irisa.fr (Matthieu Dorier)

1. Introduction

As supercomputers become larger and more complex, one critical challenge is to efficiently handle the immense amounts of data generated by extreme-scale simulations. The traditional approach to data management consists of writing data to a parallel file system, using a high-level I/O library on top of a standardized interface such as MPI-I/O. This data is then read back for analysis and visualization purpose.

One major issue posed by this traditional approach to data management is that it induces a high performance variability. This variability can be observed at different levels. Within a single application, I/O contention across processes leads to large variations in the time each process takes to complete its I/O operations (I/O jitter). Such differences from a process to another in a massively parallel application makes all processes wait for the slowest one. These processes thus waste valuable computation time. The variability is even larger from one I/O phase to another, due to interference with other applications sharing the same parallel file system.

While scientists have found a potential solution to this problem by coupling their simulations with visualization software in order to bypass data storage and derive results early on, the current practices of coupling simulations with visualization tools also expose simulations to high performance variability, as their run time does not depend anymore on their own scalability only, but also on the scalability of visualization algorithms. This particular problem is further amplified in the context of interactive in situ visualization, where the user himself and his interactions with the simulation become the cause of run-time variability.

To make an efficient use of future exascale machines, it becomes important to provide data management solutions that do not solely focus on pure performance, but address performance variability as well. Addressing this variability is indeed the key to ensure that each and every component of these future platforms is optimally used.

To address these challenges, we have proposed a new system for I/O and data management called Damaris. Damaris leverages dedicated I/O cores on each multicore SMP (Symmetric multiprocessing) node, along with the use of shared memory, to efficiently perform asynchronous data processing I/O and in situ visualization. We picked this approach based on the intuition that the usage of dedicated cores for I/O-related tasks combined with the usage of intranode shared memory can help overlapping I/O with computation, but also lowering the pressure on the storage system by reducing the number of files to be stored and, at the same time, the amount of data. Such dedicated resources can indeed perform data aggregation, filtering or compression, all in an asynchronous manner. Moreover, such dedicated cores can further be leveraged to enable non-intrusive in situ data visualization with optimized resource usage. Some of these aspects of the Damaris approach have been introduced in previous conference papers [1, 2]. This paper aims to provide a comprehensive, global presentation and discussion of the Damaris approach in its current state and of its evaluation

and applications.

We evaluated Damaris on three different platforms including the Kraken Cray XT5 supercomputer [3], with the CM1 atmospheric model [4] and the Nek5000 [5] computational fluid dynamics code. By overlapping I/O with computation and by gathering data into large files while avoiding synchronization between cores, our solution brings several benefits: (1) it fully hides the jitter as well as all I/O-related costs, which makes the simulations performance predictable; (2) it substantially increases the sustained write throughput (by a factor of 15 in CM1, 4.6 in Nek5000) compared with standard approaches; (3) it allows almost perfect scalability of the simulation (up to over 9,000 cores with CM1 on Kraken), as opposed to state-of-the-art approaches which fail to scale; (4) it enables data compression without any additional overhead, leading to a major reduction of storage requirements.

Furthermore, we extended Damaris with Damaris/Viz, an in situ visualization framework based on the Damaris approach. By leveraging dedicated cores, external high-level structure descriptions and a simple API, our framework provides adaptable in situ visualization to existing simulations at a low instrumentation cost. Results obtained with the Nek5000 and CM1 simulations show that our framework can completely hide the performance impact of visualization tasks and the resulting run-time variability. In addition, the proposed API allows efficient memory usage through a shared-memory-based, zero-copy communication model.

Finally, in order to compare the Damaris, dedicated-core-based approach with other approaches such as dedicated nodes, forwarding nodes, and staging areas, we further extended Damaris to support the use of dedicated nodes as well. We leverage again the CM1 and Nek5000 simulations on Grid'5000, the national French grid testbed, to shed light on the conditions under which a dedicated-core-based approach to I/O is more suitable than a dedicated-node-based one, and vice versa.

To the best of our knowledge, Damaris is the first open-source middleware to enable the use of dedicated cores or/and dedicated nodes for data management tasks ranging from storage I/O to complex in situ visualization scenarios.

The rest of this paper is organized as follows: Section 2 presents the background and motivation for our work, discusses the limitations of current approaches to I/O and to in situ visualization. Our Damaris approach, including its design principles, implementation detail and use cases, is described in Section 3. We evaluate Damaris in Section 4, first in scenarios related to storage I/O, then in scenarios related to in situ visualization. Our experimental evaluation continues in Section 5 with a comparison between dedicated cores and dedicated nodes in various situations. Section 6 discusses our positioning with respect to related work and Section 7 summarizes our conclusions and discusses open further directions.

2. Background and Motivation

HPC simulations create large amounts of data that are then read offline by analysis tools. In the following we present the traditional approaches to parallel I/O as well as the problems they pose in terms of performance variability. We then dive into the trend toward coupling simulations with analysis and visualization tools, going from offline to in situ analysis and visualization.

2.1. I/O and Storage for Large-Scale HPC Simulations

Two I/O approaches are commonly have been traditionally used for performing I/O in large-scale simulations.

The File-per-process approach consists of having each process access its own file. This reduces possible interference between the I/O of different processes, but increases the number of metadata operations. This is especially a problem for file systems with a single metadata server, such as Lustre [6]. It is also hard to manage the large number of files thus created and have them read by analysis or visualization codes that use a different number of processes

Collective I/O leverages communication phases between processes to aggregate access requests and reorganize them. These operations are typically used when several processes need to access different parts of a shared file, and benefit from tight interactions between the file system and the MPI-I/O layer in order to optimize the application's access pattern [7].

2.1.1. Variability in Traditional I/O Approaches

The periodic nature of scientific simulations, which alternate between computation and I/O phases, leads to burst of I/O activity. The overlap between computation and I/O is reduced, so that both the compute nodes and the I/O subsystem may be idle for periods of time.

With larger machines, the higher degree of I/O concurrency between processes of a single application or between concurrent applications pushes the I/O system to its limits. This leads to a substantial variability in I/O performance. Reducing or hiding this variability is critical, as it is an effective way to make a more efficient use of these new computing platforms through improved predictability of the behavior and of the execution time of applications.

Figure 1 illustrates this variability with the IOR application [8], a typical benchmark used to evaluate the performance of parallel file systems with predefined I/O patterns. It shows that even with very well optimized I/O (each process here writes the same amount of data contiguously in a separate file using large requests that match the file system's distribution policy) there is a large difference in the time taken by each process to complete its I/O operations within a single I/O phase and also across I/O phases. Since during these I/O phases all processes have to wait for the slowest one before resuming computation, this I/O variability leads to a waste of performance and to unpredictable overall run times. I/O variability is therefore a key issue that we aim to address in this paper.

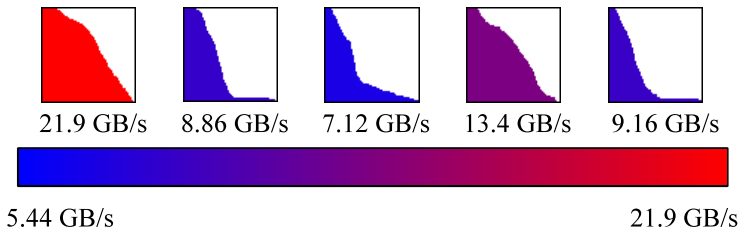


Figure 1: Variability across processes and across I/O phases in the IOR benchmark using a file-per-process approach on Grid'5000's Rennes site [9], with a PVFS2 [10] file system. Each graph represents a write phase. The 576 processes are sorted by write time on the y axis and an horizontal line is draw with a length proportional to this write time. These graphs are normalized so that the longest write time spawns the entire graph. Each graph is colored according to a scale that gives the aggregate throughput of the phase, that is, the total amount of data written divided by the write time of the slowest process.[2]

2.1.2. Causes and Effects of the I/O Variability

Skinner at al. [11] point out four causes of performance variability in supercomputers (here presented in a different order).

1. Communication, causing synchronization between processes that run within the same node or on separate nodes. In particular, network access contention causes collective algorithms to suffer from variability in point-to-point communications.
2. Kernel process scheduling, together with the jitter introduced by the operating system.
3. Resource contention within multicore nodes, caused by several cores accessing shared caches, main memory and network devices.
4. Cross-application contention, which constitutes a random variability coming from simultaneous accesses to shared components of the computing platform, such as the network or the storage system, by distinct applications.

Future systems will have additional sources of variability, such as power management, and fault masking activities. Issues 1 and 2, respectively, cause communication and computation jitter. Issue 1 can be addressed through more efficient network hardware and collective communication algorithms. The use of lightweight kernels with less support for process scheduling can alleviate issue 2. Issues 3 and 4, on the other hand, cause I/O performance variability.

At the level of a node, the increasing number of cores per node in recent machines makes it difficult for all cores to access the network all at once with an optimal throughput. Requests are serialized in network devices, leading to

[2]Due to the use of colors, this figure may not be properly interpretable if this document was printed in black and white. Please refer to an electronic version.

a different service time for each core. This problem is further amplified by the fact that an I/O phase consists of many requests that are thus serialized in an unpredictable manner.

Parallel file systems also represent a well-known bottleneck and a source of high variability [12]. The time taken by a process to write some data can vary by several orders of magnitude from one process to another and from one I/O phase to another depending on many factors, including (1) network contention when several nodes send requests to the same I/O server [13], (2) access contention at the level of the file system's metadata server(s) when many files are created simultaneously [14], (3) unpredictable parallelization of I/O requests across I/O servers due to different I/O patterns [15], (4) additional disk-head movements due to the interleaving of requests coming from different processes or applications [16]. Other source of I/O variability at disk level include the overheads of RAID group reconstruction, data scrubbing overheads, or various firmware activities.

Lofstead et al. [15] present I/O variability in terms of *interference*, with the distinction between *internal interference* caused by access contention between processes of the same application, and *external interference* that are due to sharing the access to the file system with other applications, possibly running on different clusters. While the sources of I/O performance variability are numerous and difficult to track, we can indeed observe that some of them originate from contentions within a single application, while other come from the contention between multiple applications concurrently running on the same platform. The following section describes how to tackle these two sources of contention.

2.1.3. Approaches to Mitigate the I/O Variability

While most efforts today address performance and scalability issues for specific types of workloads and software or hardware components, few efforts target the causes of performance variability. We highlight two practical ways of hiding or mitigating the I/O variability.

Asynchronous I/O. The main solution to prevent an application from being impacted by its I/O consists of using asynchronous I/O operations, i.e., non-blocking operations that proceed in the background of the computation.

The MPI 2 standard proposes rudimentary asynchronous I/O functions that aim to overlap computation with I/O. Yet these functions are available only for independent I/O operations. Besides, popular implementations of the MPI-I/O standard such as ROMIO [17] actually implement most of these functions as synchronous. Only the small set of functions that handle contiguous accesses have been made asynchronous, provided that the backend file system supports it.

Released in 2012, the MPI 3 standard completes this interface with asynchronous collective I/O primitives. Again, their actual implementation is mostly synchronous. As of today, there is no way to leverage completely asynchronous I/O using only MPI-I/O. Higher-level libraries such as HDF5 [18, 19] or NetCDF [20] have also no support yet for asynchronous I/O.

Dedicated I/O Resources. Over the past few years, dedicated I/O resources have been proposed to address the limitation of MPI implementations in terms of asynchronous I/O. These resources can take various forms. Explicit I/O threads [21] have been used to achieve fully asynchronous I/O at the potential price of additional OS jitter. Dedicated cores have been proposed to leverage a subset of cores in each multicore node used by the application [1, 22], and have them perform I/O operations on behalf of the cores that run the application. Staging areas [23, 24, 25] is another approach that usually consists of dedicated nodes deployed along with an application. Forwarding nodes [26, 27] and burst buffers [28, 29] consist of a set of nodes, independent of the applications and interposed between the compute nodes and the storage system. These nodes may feature a larger memory capacity than compute nodes, in the form of SSDs or NVRAMs.

This trend toward using dedicated resources has benefited the field of data analysis and visualization as well, where dedicated cores or nodes are seen as new ways to efficiently get access to simulations' data as they are generated. The next section explores this trend in more details.

2.2. Analysis and Visualization: an Overlooked Process

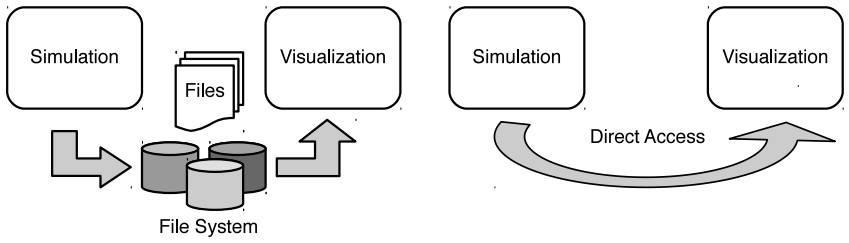
Data produced by HPC simulations can serve several purposes. One of them is fault tolerance using a checkpoint/restart method. The other, and most important, is the analysis and visualization of the simulated phenomenon. Analysis and visualization are important components of the process that leads from running a simulation to actually *discovering knowledge*.

Given the increasing computation power in recent machines and the trend toward using dedicated resources, it will become more and more common to couple the simulation with the analysis and visualization tools. Simulation/Visualization coupling consists of making the simulation send its data directly to a visualization software instead of storing it and processing it offline. This approach, termed *in situ visualization* and illustrated in Figure 2 (b), has the advantage of bypassing the storage system and producing results faster. It also allows scientists to control their simulations as they run, efficiently overlapping simulation and knowledge discovery.

2.2.1. A Taxonomy of In Situ Visualization Methods

Several in situ visualization strategies exist that we separate into two main categories –tightly coupled and loosely coupled– depending on where visualization tasks run.

Tightly-Coupled In Situ Visualization. In a tightly-coupled scenario, the analysis and visualization codes run on the same node as the simulation and share its resources. The main advantage of this scenario is the proximity to the data, which can be retrieved directly from the memory of the simulation. Its drawback lies in the impact that such analysis and visualization tasks can have on the performance of the simulation and on the variability of its run time. Within



(a) Traditional Scientific Workflow (b) Coupling Simulation/Visualization

Figure 2: Two approaches to retrieve insight from large-scale simulations: (a) the traditional approach of storing data in a parallel file system and reading it offline, (b) the new trend towards simulation/visualization coupling.

this category, we make a distinction between *time partitioning* and *space partitioning*.

Time-partitioning visualization consists of periodically stopping the simulation to perform visualization tasks. This is the most commonly used method. For example, it is implemented in VisIt's *libsim* library [30] and ParaView's *Catalyst* library [31, 32].

In a space-partitioning mode, dedicated cores perform visualization in parallel with the simulation. This mode poses challenges in efficiently sharing data between the cores running the simulation and the cores running the visualization tasks, as these tasks progress in parallel. It also reduces the number of cores available to the simulation.

Loosely-Coupled In Situ Visualization. In a loosely coupled scenario, analysis and visualization codes run on a separate set of resources, that is, a separate set of nodes located either in the same supercomputer as the simulation [33, 34], or in a remote cluster [35]. The data is sent from the simulation to the visualization nodes through the network.

Some in situ visualization frameworks such as GLEAN [36] can be considered hybrid, placing some tasks close to the simulation in a time-partitioning manner while other tasks run on dedicated nodes.

2.2.2. From Offline to In Situ Visualization: Another Source of Variability

The increasing amounts of data generated by scientific simulations also leads to performance degradations when it comes to reading back data for analysis and visualization [37, 38]. While I/O introduces run time variability, in situ analysis and visualization can also negatively impact the performance of the simulation/visualization complete workflow. For instance, periodically stopping the simulation to perform in situ visualization in a time-partitioning manner leads to a loss of performance and an increase of run-time variability. Contrary to the performance of the simulation itself, the performance of visualization

tasks may depend on the content of the data and is therefore unbalanced across processes and across iterations. This variability is further amplified if the in situ visualization framework is interactive, in which case the user himself impacts the performance of his application.

In a loosely-coupled approach to in situ visualization, sending data through the network potentially impacts the performance of the simulation and forces a reduced number of nodes to sustain the input of a large amount of data. Transferring such large amounts of data through the network also have a potentially larger impact on the simulation than running visualization tasks in a tightly-coupled manner.

2.3. Our Vision: Using Dedicated Cores for I/O and In Situ Visualization

Despite the limitations of the traditional, offline approach to data analysis and visualization, users are still seldom moving to purely in situ visualization and analysis [39, 40, 41]. The first reason is the development cost of such a step in large codes that were maintained for decades. The second reason is that storage I/O is still required for checkpoint-based fault tolerance, which makes offline analysis of checkpoints the natural candidate for scientific discovery.

To push further the adoption of in situ visualization and increase the productivity of the overall scientific workflow, *we postulate that a framework should be provided that deals with all aspects of Big Data management in HPC simulations*, including efficient I/O but also in situ processing, analysis and visualization of the produced data. Such a framework can at the same time provide efficient storage I/O for data that need to be stored, and efficient in situ visualization to speed up knowledge discovery and enable simulation monitoring.

Over the past 4 years we have been addressing this challenge by proposing, designing and implementing the Damaris system to data management. Damaris proposes to dedicate cores in multicore nodes for any type of data management task, including I/O and in situ visualization. We tried to make Damaris *simple to use*, *flexible*, *portable* and *efficient* in order to ease its adoption by the HPC community. The following section gives an overview of this approach and its implementation.

3. The Damaris Approach: an Overview

In order to address both I/O and in situ analysis/visualization issues, we propose to gather the I/O operations into a set of dedicated cores in each multicore node. These cores (typically one per node) are dedicated to data management tasks (i.e., they do not run the simulation code) in order to overlap writes and analysis tasks with computation and avoid contention for accesses to the file system. The cores running the simulation and the dedicated cores communicate data through shared memory. We call this approach Damaris. Its design, implementation and API are described below.

3.1. Design Principles

The Damaris approach is based on four main design principles.

3.1.1. Dedicated Cores

The Damaris approach is based on a set of processes running on dedicated cores in every multicore node. Each dedicated core performs in situ processing and I/O in response to user-defined events sent by the simulation. We call a process running the simulation a *client*, and a process running on a dedicated core a *server*. One important aspect of Damaris is that dedicated cores do not run the simulation. With the current trend in hardware solutions, the number of cores per node increases. Thus dedicating one or a few cores has a diminishing impact on the performance of the simulation. Hence, our approach primarily targets SMP nodes featuring a large number of cores per node: 12 to 24 in our experiments. This arrangement might be even more beneficial in future systems, for a variety of reasons: The number of cores increasing, neither memory bandwidth nor power constraints may allow all cores to run compute-intensive code; and reduced switching between different types of executions improves performance.

3.1.2. Data Transfers through Shared Memory

Damaris handles large data transfers from clients to servers through shared memory. This makes a write as fast as a `memcpy` and also enables direct allocation of variables within the shared memory. This option is especially useful to reduce the memory requirements of in situ visualization tasks, which can directly access the memory of the simulation without requiring a copy (see our previous work [2]).

3.1.3. High-Level Data Abstraction

Clients write enriched datasets in a way similar to scientific I/O libraries such as HDF5 [19] or NetCDF [20]. That is, the data output by the simulation is organized into a hierarchy of groups and variables, with additional metadata such as the description of variables, their type, unit, and layout in memory. The dedicated cores thus have enough knowledge of incoming datasets to write them in existing high-level formats. This design principle differs from other approaches that capture I/O operations at a lower level [22, 29]. These approaches indeed lose the semantics of the data being written. While our design choice forces us to modify the simulation so that it writes its data using Damaris' API, is allows to implement semantic-aware data processing functionalities in dedicated cores. In particular, keeping this level of semantics is mandatory in order for dedicated cores to be able to write data in a standard, high-level format such as HDF5 or NetCDF, or to feed an in situ visualization pipeline.

3.1.4. Extensibility through Plugins

Servers can perform data transformations prior to writing them, as well as analysis and visualization. One major design principle in the Damaris approach is the possibility for users to provide these transformations through a plugin system, thus adapting Damaris to the particular requirements of their application. Implementing such a plugin system at a lower level would not be possible,

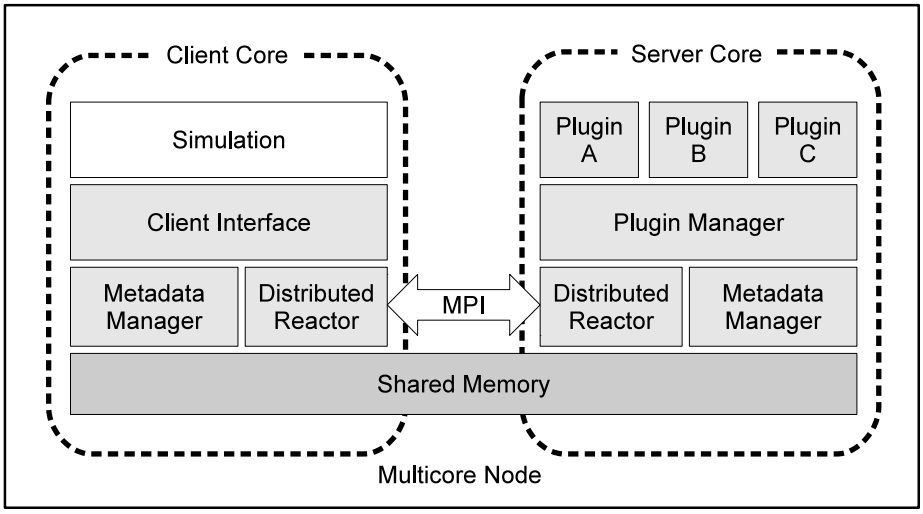


Figure 3: Software architecture of the implementation of Damaris.

as it would not have access to the high-level information about the data (e.g., dimensions of arrays, data types, physical meaning of the variable within the simulation, etc.).

3.2. Architecture

Figure 3 presents the software architecture underlying the Damaris approach. While Damaris can dedicate several cores in large multicore nodes, only one client and one server are represented here.

Damaris has been designed in a highly modular way and features a number of decoupled, reusable software components. The *Shared Memory* component handles the shared buffer and ensures the safety of concurrent allocations/deallocations. The *Distributed Reactor* handles communications between clients and servers, and across servers. The *Metadata Manager* stores high-level information related to the data being transferred (type, size, layout, etc.). Finally the *Plugin Manager* on the server side loads and runs user-provided plugins.

This modular architecture greatly simplified the adaptation to several HPC platforms and simulations, as well as the development of extensions to support various scenarios such as storage, in situ visualization, data compression or I/O scheduling. The following sections describe each component in more detail.

3.2.1. Shared Memory

Data communications between the clients and the servers are performed through the Shared Memory component. A large memory buffer is created on each node by the dedicated cores at start time, with a size set by the user (typically several MB to several GB). Thus the user has full control over the

resources allocated to Damaris. When a client submits new data, it reserves a segment of this shared-memory buffer. It then copies its data using the returned pointer so that the local memory can be reused.

3.2.2. Distributed Reactor

The Distributed Reactor is the most complex component of Damaris. It builds on the *Reactor* design pattern [42] to provide the means by which different cores (clients and servers) communicate through MPI. Reactor is a behavioral pattern that handles requests concurrently sent to an application by one or more clients. The Reactor asynchronously listens to a set of *channels* connecting it to its clients. The clients send small events that are associated with event handlers (i.e., functions) in the Reactor. A synchronous event demultiplexer is in charge of queuing the events received by the Reactor and calling the appropriate event handlers. While clients communicate data through shared memory, they use the Distributed Reactor, based on MPI, to send short notifications that either new data is available in shared memory, or that a plugin should be triggered.

Contrary to a normal Reactor design pattern (as used in *Boost.ASIO*[3] for example), our Distributed Reactor also provides elaborate collective operations.

Asynchronous atomic multicast: A process can broadcast an event to a group of processes at once. This operation is asynchronous, that is, the sender does not wait for the event to be processed by all receivers to resume its activity. A receiver only processes the event when all other receivers are ready to process it as well. It is also atomic, that is, if two distinct processes broadcast a different event, the Distributed Reactor ensures that all receivers will handle the two events in the same order.

Asynchronous atomic labeled barrier: We call a "labeled" barrier across a set of processes a synchronization barrier associated with an event (its label). After all processes reach the barrier, they all invoke the event handler associated with the event. This ensures that all processes agree to execute the same code at the same *logical* time. This primitive is asynchronous: it borrows its semantics from MPI 3's `MPI_Ibarrier` non-blocking barrier. It is atomic according to the same definition as the asynchronous atomic multicast.

These two distributed algorithms are important in the design of in situ processing tasks that include communications between servers. In particular, they ensure that plugins will be triggered in the same order in all servers, allowing collective communications to safely take place within these plugins.

3.2.3. Metadata Manager

The Metadata Manager component keeps information related to the data being written, including *variables*, *layouts* (describing the type and shape of

[3] See `http://www.boost.org/`

blocks of data), *parameters*, etc. It is initialized using an XML configuration file.

This design principle is inspired by ADIOS [43] and other tools such as EPSN [44]. In traditional data formats such as HDF5, several functions have to be called by the simulation to provide metadata information prior to actually writing data. The use of an XML file in Damaris presents several advantages. First, the description of data provided by the configuration file can be changed without changing the simulation itself, and the amount of code required to use Damaris in a simulation is reduced compared to existing data formats. Second, it prevents clients from transferring metadata to dedicated cores through shared memory. Clients communicate only data along with the minimum information required by dedicated cores to retrieve the full description in their own Metadata Manager.

Contrary to the XDMF format [45], which leverages XML to store scientific datasets along with metadata (or points to data in external HDF5 files), our XML file only provides metadata related to data produced by the simulation. It is not intended to be an output format, or become part of one.

3.2.4. Plugin Manager

The Plugin Manager is the component that loads and stores plugins. Plugins are pieces of C++ or Python codes provided by the user. The Plugin Manager is capable of loading functions from dynamic libraries or scripts as well as from the simulation's code itself. It is initialized from the XML configuration file. Again, the use of a common configuration file between clients and servers allows different processes to refer to the same plugin through an identifier rather than its full name and attributes.

A server can call a plugin when it receives its corresponding event, or wait for all clients in a node or in the entire simulation to have sent the event. In these later cases, the collective algorithms provided by the Distributed Reactor ensure that all servers call the plugins in the same order.

3.3. Implementation

The Damaris approach is intended to be the basis for a generic, platform-independent, application-independent, easy-to-use tool. This section describes its main API and provides some technical details of its implementation.

3.3.1. Client API

Our implementation provides client-side interfaces for C, C++ and Fortran applications written with MPI. This API can be summarized by the following functions.

- `damaris_initialize("config.xml")` initializes the resources used by Damaris using the configuration file given as parameter. All cores (clients and servers) call this function at the beginning of the simulation.

- `damaris_start()` is called by all cores to start servers on dedicated cores. The servers block within this function, while the clients return and proceed with the simulation.

- `damaris_get_client_comm()` provides an MPI communicator gathering only clients. Indeed the MPI_COMM_WORLD communicator contains both clients and servers and cannot be used by the simulation anymore for its collective communications.

- `damaris_write("var_name",data)` is called by clients. It copies the data in shared memory along with minimal information and notifies the server on the same node. All additional information such as the size of the data and its layout can be found by the servers in the configuration file.

- `damaris_alloc("variable")` is similar to *malloc* (or *allocate* in Fortran, *new* in C++). It is called by clients to allocate a portion of shared memory to hold the variable for a given iteration and returns a pointer. Only the simulation is aware of this allocation, dedicated cores cannot access the data. The returned buffer is expected to be used as output buffer for the next iteration.

- `damaris_commit("variable")` is called by clients when the simulation has finished writing to the current buffer associated with the variable. It sends the location of the data to the dedicated cores. Both the simulation and dedicated cores can read the data. At this point, clients will use the buffer as input for the next iteration while dedicated cores will use it as input for visualization tasks.

- `damaris_clear("variable")` is called by clients to notify the dedicated cores that the committed data for this variable will no longer be used by the simulation. It can safely be processed, stored or removed from shared memory. The clients will issue another `damaris_alloc` to get a new portion of shared memory to use as output buffer.

- `damaris_signal("event_name")` is called by clients to send a custom event to the server in order to trigger a plugin predefined in the configuration file.

- `damaris_end_iteration()` notifies the servers that the simulation has reached the end of an iteration and will start the next one. This allows dedicated cores to know that the data written in shared memory is consistent and nothing more should be expected for this iteration.

- `damaris_stop()` stops the servers on dedicated cores, making them leave the `damaris_start` function.

- `damaris_finalize()` frees the resources used by Damaris. It is called by all processes after servers have been stopped on dedicated cores (using `damaris_stop`) before terminating the simulation.

3.3.2. Technical Implementation Details

Damaris leverages the *Boost.Interprocess* library[4] to implement several versions of the Shared Memory component, suitable for different platforms.

Our implementation of the Distributed Reactor relies on MPI 2 communication primitives and, in particular, non-blocking *send* and *receive* operations. Events are simply implemented as 0-byte messages with the MPI *tag* carrying the type of the event. Since the MPI 3 standard provides new non-blocking collective functions such as `MPI_Ireduce` or `MPI_Ibarrier`, our Distributed Reactor could be easily re-implemented with these MPI 3 functions without any impact on the rest of Damaris' implementation.

Finally we used Model-Driven Engineering (MDE) techniques to implement the Metadata Manager. Most of the source code of the Metadata Manager is indeed automatically generated from an XSD metamodel. This metamodel describes the concepts of *variables*, *layouts*, etc. as well as their relations to one another and how they are described in an XML format. The XSD file is used to synthesize C++ classes that correspond to the metamodel.

3.4. Managing Data with Damaris

Damaris is not a data format. It only provides a framework to dedicate cores for custom data processing and I/O tasks, to transfer data through shared memory and to call plugins. Thanks to its plugin system, Damaris can be adapted to many scenarios of in situ data processing. In this paper, we specifically use it to periodically write data and to perform in situ visualization.

3.4.1. Writing Data

We implemented a plugin that gathers data from client cores and writes them into HDF5 files. Each server running on a dedicated core produces a single file per iteration. Compared with the file-per-process approach, this way of writing produces fewer, bigger files, thus mitigating the bottleneck in metadata servers when files are created. Writing from a reduced number of writers also has the advantage of limiting network access contention across the cores of the same node. Finally, issuing bigger writes to the file system usually allows for better performance. Compared with the collective I/O approach, our writer plugin does not require synchronization between processes.

3.4.2. Visualizing and Analyzing

The high-level data description provided by Damaris enables a connection with existing visualization and analysis packages, including VisIt [46] or ParaView [47], in order to build a full in situ visualization framework. Both VisIt and ParaView perform in situ visualization from in-memory data. Given that each of these software has strengths, a major advantage of our approach is the ability to switch between them with no code modification in the simulation.

[4] See `http://www.boost.org/`

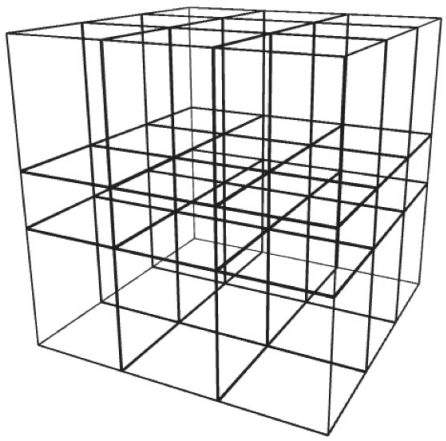


Figure 4: Example of a $4 \times 4 \times 4$ rectilinear grid described by three arrays of coordinates. In this example there is a scalar value (such as *temperature* or *wind velocity*) at each node. The mesh itself is described through three coordinate arrays: `mesh_x = {0.0,1.0,2.0,3.0}; mesh_y = {0.0,1.0,2.0,3.0}; mesh_z = {0.0,1.2,1.8,3.0}`.

We leveraged the XSD-based metadata management in Damaris to provide the necessary information to bridge simulations to existing visualization software. By investigating the in situ interfaces of different visualization packages including ParaView, VisIt, ezViz [48] and VTK [49], we devised a generic description of visualizable structures such as meshes, points or curves. Additionally, the Distributed Reactor enables synchronization between dedicated cores, which is necessary to run the parallel rendering algorithms implemented by the aforementioned visualization software.

3.5. Code Sample using Damaris

Listing 1 is an example of a Fortran program that makes use of Damaris. It writes three 1D arrays representing the coordinates of a rectilinear mesh. At every iteration it then writes a 3D array representing temperature values on the points of the mesh and sends an event to the dedicated core. The associated configuration file, shown in Listing 2, describes the data that is expected to be received by the servers, and the action to perform upon reception of the event. More specifically, lines 14, 15, 16 and 18 of this XML file define *layouts*, which describe the type and dimensions of a piece of data. Lines 26 to 33 define a group, and within this group a set of variables that use these layouts. The temperature variable is defined in line 35. Finally line 38 associates an event with a function (or *action*) to be called when the event is received. It also locates the function within a dynamically-loaded library.

The configuration file also contains information for visualization software. Lines 20 to 24 in the XML file correspond to mesh structure drawn in Figure 4, and built from the three coordinate variables. The temperature variable is mapped onto this mesh using its *mesh* attribute.

```
1  program example
2    integer ierr, is_client
3    real, dimension(64,16,2) :: temperature
4    real, dimension(4) :: x3d, y3d, z3d
5
6    ! initialization
7    call damaris_initialize_f("config.xml", MPI_COMM_WORLD, ierr)
8    call damaris_start_f(is_client, ierr)
9
10   if(is_client.eq.1) then
11
12        ! writing non-time-varying data
13     call damaris_write_f("coordinates/x3d", x3d, ierr)
14     call damaris_write_f("coordinates/y3d", y3d, ierr)
15     call damaris_write_f("coordinates/z3d", z3d, ierr)
16
17     do while(...) ! simulation main loop
18       ...
19           ! writing temperature data
20       call damaris_write_f("temperature", temperature, ierr)
21           ! sending signal
22       call damaris_signal_f("my_event", ierr)
23           ! end of iteration
24       call damaris_end_iteration_f(ierr)
25       ...
26     enddo
27     ! stopping the servers
28     call damaris_stop_f(ierr)
29   endif
30
31   ! finalization
32   call damaris_finalize_f(ierr)
33 end program example
```

Listing 1: Example of Fortran simulation using Damaris.

4. Evaluation

We evaluated Damaris with the CM1 atmospheric simulation [4] and ANL's Nek5000 CFD solver [5], on several platforms: NICS's Kraken [3], three clusters of the French Grid'5000 platform [9], NCSA's BluePrint cluster and the Blue Waters supercomputer [50]. In the following, we first evaluate Damaris in the context of improving I/O performance by hiding the I/O variability. We then evaluate the use of Damaris for several other data management tasks, including data compression, I/O scheduling and in situ visualization.

4.1. Addressing the I/O Bottleneck with Damaris

In this first evaluation part, we show how Damaris is used to improve I/O performance.

4.1.1. Description of the Applications

The following applications were used in our experiments.

CM1 (Cloud Model 1) is used for atmospheric research and is suitable for modeling small-scale atmospheric phenomena such as thunderstorms and

```
<simulation name="my_simulation" language="c"
    xmlns="http://damaris.gforge.inria.fr/damaris/model">
    <architecture>
      <domains count="1"/>
      <dedicated cores="1"/>
      <buffer name="the_buffer" size="67108864" />
      <queue name="the_queue" size="100" />
    </architecture>
    <data>
       <parameter name="w" type="int" value="4" />
       <parameter name="h" type="int" value="4" />
       <parameter name="d" type="int" value="4" />

       <layout name="mesh_x_layout" type="float" dimensions="w" />
       <layout name="mesh_y_layout" type="float" dimensions="h" />
       <layout name="mesh_z_layout" type="float" dimensions="d" />

       <layout name="data_layout" type="double" dimensions="w,h,d"/>

       <mesh name="mesh3d" type="rectilinear" topology="3">
           <coord name="coordinates/x3d" unit="m" label="Width"/>
           <coord name="coordinates/y3d" unit="m" label="Height"/>
           <coord name="coordinates/z3d" unit="m" label="Depth"/>
       </mesh>

       <group name="coordinates">
           <variable name="x3d" layout="mesh_x_layout"
               visualizable="false" time-varying="false" />
           <variable name="y3d" layout="mesh_y_layout"
               visualizable="false" time-varying="false" />
           <variable name="z3d" layout="mesh_z_layout"
               visualizable="false" time-varying="false" />
       </group>

       <variable name="temperature" layout="data_layout" mesh="mesh3d"/>
    </data>
    <actions>
      <event name="my_event" action="my_function" using="my_plugin.so" />
    </actions>
</simulation>
```

Listing 2: Configuration file associated with the Fortran example.

tornadoes. It follows a typical behavior of scientific simulations, which alternate computation phases and I/O phases. The simulated domain is a regular 3D grid representing part of the atmosphere. Each point in this domain is characterized by a set of variables such as local *temperature* or *wind speed*. CM1 is written in Fortran 90. Parallelization is done using MPI, by distributing the 3D array along a 2D grid of equally-sized subdomains, each of which is handled by a process. The I/O phase leverages either HDF5 to write one file per process, or pHDF5 [51] to write in a shared file in a collective manner. One of the advantages of using a file-per-process approach is that compression can be enabled, which cannot be done with pHDF5. However, at large process counts, the file-per-process approach generates a large number of files, making all subsequent analysis tasks intractable.

Nek5000 is a computational fluid dynamics solver based on the spectral element method. It is actively developed at ANL's Mathematics and Computer Science Division. It is written in Fortran 77 and solves its governing equations on an unstructured mesh. This mesh consists of multiple elements distributed across processes; each element is a small curvilinear mesh. Each point of the mesh carries the three components of the fluid's local velocity, as well as other variables. We chose Nek5000 for this particular meshing structure, different from CM1, and for the fact that it is substantially more memory-hungry than CM1. We modified Nek5000 in order to pass the mesh elements and fields data to Damaris.

Nek5000 takes as input the mesh on which to solve the equations, along with initial conditions. We call this set a *configuration*. In our experimental evaluation, we used the *MATiS* configuration, which was designed to run on 512 to 2048 cores. Another configuration, *turbChannel*, is used in Section 4.2 to evaluate in situ visualization. This configuration was designed to run on 32 to 64 cores.

4.1.2. Platforms and Configurations

With the CM1 application, our goal was to optimize CM1's I/O for future use on the upcoming Blue Waters Petascale supercomputer. Therefore we started with NCSA's IBM Power5 BluePrint platform as it was supposed to be representative of Blue Waters' hardware. On this platform, we evaluated the scalability of the CM1 application with respect to the size of its output, with the file-per-process and Damaris approaches. We then experimented on the *parapluie* cluster of Grid'5000's Rennes site. This cluster features 24-core nodes, which makes it very suitable to our approach based on dedicated cores. We then moved our experiments to NICS's Kraken supercomputer, which, in addition to allowing runs at much larger scales, has a hardware configuration very close to that of Blue Waters' final design.

With Nek5000, our goal was to confirm the usability of Damaris with a more memory-hungry application. We completed our experimentation on the *stremi* cluster of Grid'5000's Reims site, which provides the same type of hardware as the *parapluie* cluster, but a different network. All these platforms are detailed hereafter, along with the configuration of CM1 and Nek5000 we used.

BluePrint is a test platform used at NCSA until 2011 when IBM was still in charge of delivering the Blue Waters supercomputer.[5] BluePrint features 120 Power5 nodes. Each node consists of 16 cores and includes 64 GB of memory. As for its file system, GPFS is deployed on 2 I/O servers. CM1 was run on 64 nodes (1024 cores), with a $960 \times 960 \times 300$-point domain. Each core handles a $30 \times 30 \times 300$-point subdomain with the

[5] As IBM terminated its contract with NCSA in 2011 and Blue Waters was finally delivered by Cray, BluePrint was later decommissioned and replaced with a test platform, JYC, matching the new Blue Waters' design.

standard approaches, that is, when no dedicated cores are used. When dedicating one core out of 16 on each node, computation cores handle a $24 \times 40 \times 300$-point subdomain. On this platform we vary the number of variables that CM1 writes, resulting in different sizes of the output. We enabled the compression feature of HDF5 for all the experiments done on this platform.

Grid'5000 is a French grid testbed. We use its *parapluie* cluster on the Rennes site and its *stremi* cluster on the Reims site. On the Rennes site, the parapluie cluster featured 40 nodes of 2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM. We run CM1 on 28 nodes (672 cores) and 38 nodes (912 cores). We deployed a PVFS file system on 15 separate I/O servers (2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk). Each PVFS node was used both as I/O server and metadata server. All nodes (including the file system's) communicate through a 20G InfiniBand 4x QDR link connected to a common Voltaire switch. We use MPICH [52] with ROMIO [53] compiled against the PVFS library, on a Debian Linux operating system. The total domain size in CM1 is $1104 \times 1120 \times 200$ points, so each core handles a $46 \times 40 \times 200$-point subdomain with a standard approach, and a $48 \times 40 \times 200$-point subdomain when one core out of 24 is used by Damaris.

On the Reims site the *stremi* cluster features the same type of node as the *parapluie* cluster. We run Nek5000 on 30 nodes (720 cores). We deploy PVFS on 4 nodes of the same cluster. Each PVFS node is used both as I/O server and metadata server. All nodes communicate through a 1G Ethernet network. We use the MATiS configuration of Nek5000, which contains 695454 elements (small $4 \times 2 \times 4$ curvilinear sub-meshes). These elements are distributed across available simulation processes. Thus the total number of elements (and thus the total amount of data output) does not vary whether we use dedicated cores or not. When no dedicated cores are used, each core handles 965 or 966 such elements. When dedicating one core out of 24, each simulation core handles 1007 or 1008 elements.

Kraken was a supercomputer deployed at the National Institute for Computational Sciences (NICS). It was ranked 11^{th} in the Top500 [54] at the time of the experiments, with a peak Linpack performance of 919.1 Teraflops. It featured 9408 Cray XT5 compute nodes connected through a Cray SeaStar2+ interconnect and running Cray Linux Environment (CLE). Each node has 12 cores and 16 GB of local memory. Kraken provided a Lustre file system using 336 block storage devices managed by 48 I/O servers and one metadata server.

On this platform, we studied the weak scalability of the file-per-process, collective I/O and Damaris approaches in CM1, that is, we measured how the run time varies with a fixed amount of data per node. When all cores in each node are used by the simulation, each client process handles a $44 \times 44 \times 200$-point subdomain. Using Damaris, each client process (11

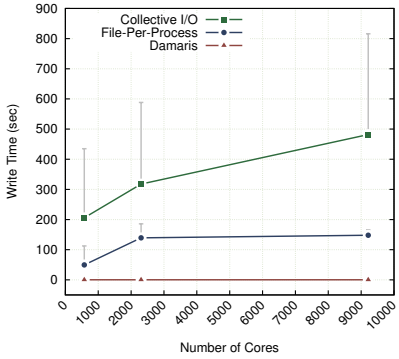


Figure 5: Duration of a write phase on Kraken (average and maximum). For readability reasons we do not plot the minimum write time. Damaris shows to completely remove the I/O variability while file-per-process and collective-I/O have a big impact on the run-time predictability.

per node) handles a $48 \times 44 \times 200$-point subdomain, which makes the total problem size equivalent for a given total number of cores.

4.1.3. How Damaris Affects the I/O Variability

Impact of the Number of Cores on the I/O Variability. We studied the impact of the number of cores on the simulation's write time with the three I/O approaches: file-per-process, collective I/O, and Damaris. To do so, we ran CM1 on Kraken with 576, 2304 and 9216 cores.

Figure 5 shows the average and maximum duration of an I/O phase on Kraken from the point of view of the simulation. It corresponds to the time between the two barriers delimiting the I/O phase. This time is extremely high and variable with Collective I/O, achieving more than 800 seconds on 9216 cores. The average of 481 seconds still represents about 70% of the overall simulation's run time.

By setting the stripe size to 32 MB instead of 1 MB in Lustre, the write time went up to 1600 seconds with a collective I/O approach. This shows that bad choices of file system's configuration can lead to extremely poor I/O performance. Yet it is hard to know in advance the configuration of the file system and I/O libraries that will lead to a good performance.

The file-per-process approach appears to lead to a lower variability, especially at large process count, and better performance than collective I/O. Yet it still represents an unpredictability (difference between the fastest and the slowest phase) of about ± 17 seconds. For a one month run, writing every 2 minutes would lead to an uncertainty of several hours to several days of run time.

When using Damaris, we dedicate one core out of 12 on each node, thus potentially reducing the computation performance for the benefit of I/O efficiency (the impact on overall application performance is discussed in the next section). As a means to reduce the I/O variability, this approach is clearly effective: the time to write from the point of view of the simulation is cut down to the time required to perform a series of copies in shared memory. It leads to an apparent write time of 0.2 seconds (as opposed to the 481 seconds of collective I/O!) and does not depend anymore on the number of processes. The variability is in order of ± 0.1 seconds (too small to be seen on the figure).

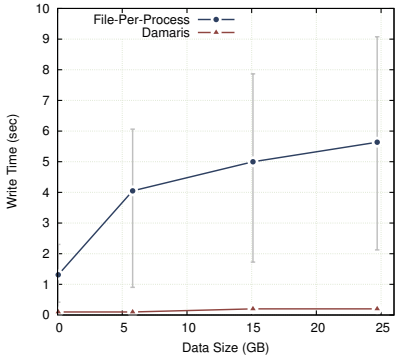


Figure 6: Duration of a write phase (average, maximum and minimum) using file-per-process and Damaris on BluePrint (1024 cores). The amount of data is given in total per write phase.

Impact of the Amount of Data on the I/O Variability. On BluePrint, we vary the amount of data. We aim to compare the file-per-process approach with Damaris with respect to different output sizes. The results are reported in Figure 6. As we increase the amount of data, the variability of the I/O time increases with the file-per-process approach. With Damaris however, the write time remains in the order of 0.2 seconds for the largest amount of data and the variability in the order of ± 0.1 seconds again.

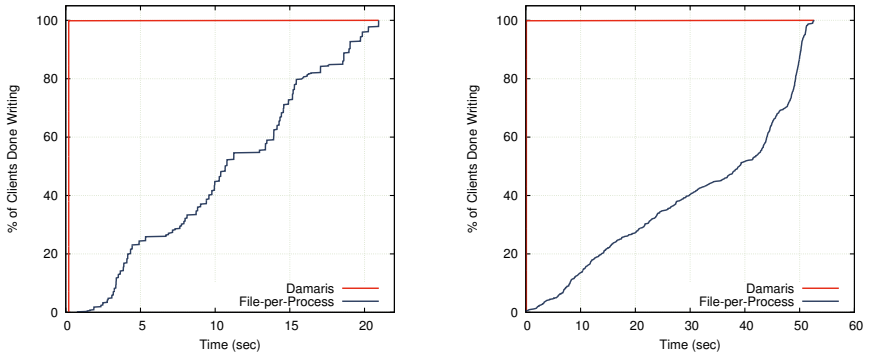
Note that on this platform, data compression was enabled. Thus the observed variability comes not only from the bottleneck at the file system level, but also from the different amounts of data that are written across processes and across iterations. This illustrates the fact that I/O variability does not only comes from the variability of performance of data transfers and storage, but also on any pre-processing task occurring before the actual I/O. Damaris is therefore able to hide this pre-processing variability as well.

Impact of the Hardware. We studied the impact of the hardware on the I/O variability using Grid'5000's *parapluie* and *stremi* clusters. With the large number of cores per node (24) in these clusters as well as a network has substantially lower performance than that of Kraken and BluePrint, we aim to illustrate the large variation of write time across cores for a single write phase.

We ran CM1 using 672 cores on the *parapluie* cluster, writing a total of 15.8 GB uncompressed data (about 24 MB per process) every 20 iterations. With the file-per-process approach, CM1 reported spending 4.22% of its time in I/O phases. Yet the fastest processes usually terminate their I/O in less than 1 second, while the slowest take more than 25 seconds. Figure 7 (a) shows the CDF (cumulative distribution function) of write times for one of these write phases, with a file-per-process approach and with Damaris.

Finally we ran Nek5000 using 720 cores on the *stremi*, writing a total of 3.5 GB per iteration using a file-per-process approach. Figure 7 (b) shows the cumulative distribution function of write time for one of these write phases with the file-per-process approach and with Damaris.

In both simulations, we observe a large difference in write time between the fastest and the slowest process with a file-per-process approach, due to access



(a) CM1 on Grid'5000 Rennes cluster (b) Nek5000 on Grid'5000 Reims cluster

Figure 7: Cumulative distribution function of the write time across processes when running CM1 on 672 cores of Grid'5000's Rennes cluster and Nek5000 on 720 cores of the Reims cluster.

contention either at the level of the network or within the file system. With Damaris however, all processes complete their write at the same time. This is due to the absence of contention when writing in shared memory.

Conclusion. Our experiments show that by replacing write phases with simple copies in shared memory and by leaving the task of performing actual I/O to dedicated cores, Damaris is able to completely hide the I/O variability from the point of view of the simulation, making the application run time more predictable.

4.1.4. Application's Scalability and I/O Overlap

Impact of Damaris on the Scalability of CM1. CM1 exhibits a very good weak scalability and very stable performance when it does not perform any I/O. Thus, as we increase the number of cores, the scalability becomes mainly driven by the scalability of the I/O phases.

Figure 8 shows the application run time for 50 iterations plus one write phase. The steady run time when no writes are performed illustrate this perfect scalability. Damaris enables a nearly perfect scalability where other approaches fail to scale. In particular, going from Collective I/O to Damaris leads to a 3.5× speedup on 9216 cores.

I/O Overhead. Another way of analyzing the effect of dedicating cores to I/O is by looking at the CPU hours wasted in I/O tasks. With a time-partitioning approach, this overhead corresponds to the duration of a write phase (expressed in hours) multiplied by the total number of cores. With dedicated cores, this overhead corresponds to the duration of the computation phase multiplied by the number of dedicated cores. Note that this metrics does not take into account

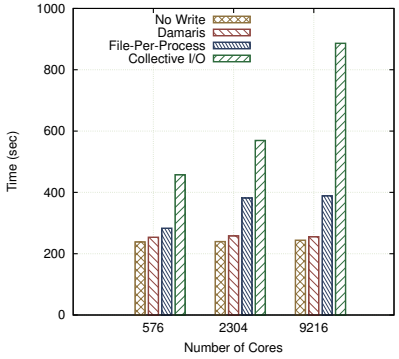


Figure 8: Average overall run time of the CM1 simulation for 50 iterations and 1 write phase on Kraken.

Cores	*Simulation w/o I/O*	File-per-process	Collective-I/O	**Damaris**
576	*38.1*	7.9	32.9	**3.4**
2304	*152.5*	89.2	203.3	**13.8**
9216	*609.8*	378.8	1244.3	**54.5**

Table 1: CPU hours wasted in I/O tasks (including processes remaining idle waiting for dependent tasks to complete), for 50 computation steps and 1 I/O phase of the CM1 application on Kraken. The "Simulation w/o I/O" column represents the CPU-hours required by the simulation to complete the 50 computation steps at this scale.

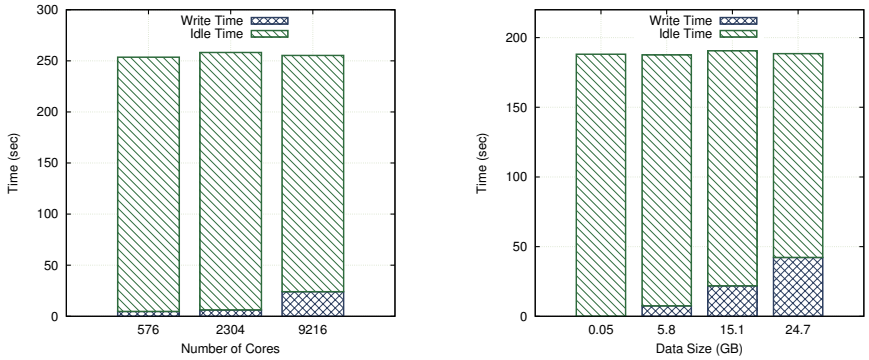
the effect of dedicating cores on the duration of a computation phase, hence the need for the study of the impact on the application's scalability, conducted earlier.

Table 1 shows the CPU hours wasted in I/O tasks, when running CM1 for 50 computation steps and 1 I/O phase. To put these numbers in perspective, the "Simulation without I/O" column shows the CPU hours required by the simulation to complete the 50 iterations without any I/O and without any dedicated cores. It shows, for example, that using a Collective-I/O approach on 9216 wastes 1244.3 CPU-hours, twice as much as the CPU-hours required by the simulation at this scale. The CPU-hours wasted by Damaris at this scale, on the other hand are as low as 54.5.

Idle Time in Damaris. Since the scalability of our approach comes from the fact that I/O overlaps with computation, we still need to show that the dedicated cores have enough time to perform the actual I/O while computation goes on.

Figure 9 shows the time used by the dedicated cores to perform the I/O on Kraken and BluePrint with CM1, as well as the time they remain idle, waiting for the next iteration to complete.

As the amount of data on each node is the same, the only explanation for the dedicated cores to take more time at larger process counts on Kraken is the access contention for the file system. On BluePrint the number of processes is constant for each experiment, thus the differences in write time come from the different amounts of data. In all configurations, our experiments show that



(a) Write / Idle Time on Kraken

(b) Write / Idle Time on BluePrint

Figure 9: Time spent by the dedicated cores writing data for each iteration. The spare time is the time dedicated cores are not performing any task.

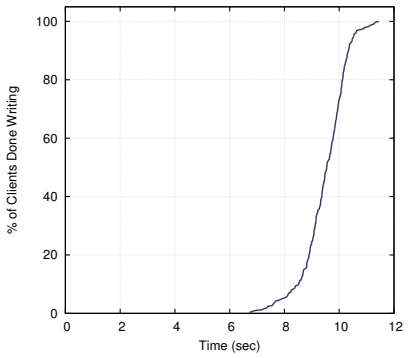


Figure 10: CDF of the time spent by dedicated cores writing (statistics across 11 iterations for 30 dedicated cores), with Nek5000 on the Reims cluster of Grid'5000.

Damaris has much spare time, during which dedicated cores remain idle. Similar results were obtained on Grid'5000. While the idle time of the dedicated cores may seem to be a waste (provided that no in situ data processing leverages it), it can reduce the energy consumption of the node; this saving will be significant in future systems that will have sophisticated dynamic power management.

With Nek5000, Figure 10 shows the cumulative distribution function (CDF) of the time spent by dedicated cores writing. This time averages to 9.41 seconds, which represents 10% of overall run time. Thus, dedicated cores remain idle 90% of the time. Additionally, this figure shows that the time spent by dedicated cores writing is stable across iterations and across processes, with a standard deviation of 1.08 seconds. This stability allows to add additional data processing tasks without worrying about the possibility that dedicated cores spend an unpredictable time writing.

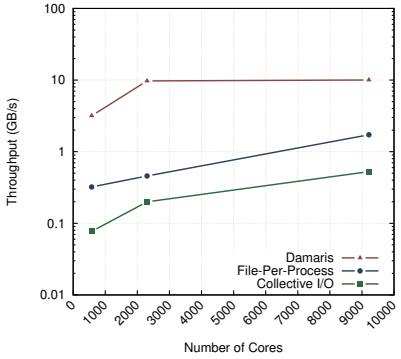


Figure 11: Average aggregate throughput achieved on Kraken with the different approaches. Damaris shows a 6 times improvement over the file-per-process approach and 15 times over Collective I/O on 9216 cores.

	Aggregate throughput
File-per-process	695 MB/s
Collective-I/O	636 MB/s
Damaris	4.32 GB/s

Table 2: Average aggregate throughput on Grid'5000's *parapluie* cluster, with CM1 running on 672 cores.

Conclusion. On all platforms, Damaris shows that it can fully overlap writes with computation and still remain idle 75% to 99% of time with CM1 (see Figure 9), and 90% with Nek5000 (see Figure 10). Thus, without impacting the application, it becomes possible to further increase the frequency of output, or to perform additional data processing operations such as in situ data analysis and visualization.

4.1.5. Aggregate I/O Throughput

We then studied the effect of Damaris on the aggregate throughput observed from the computation nodes to the file system, that is, the total amount of data output by the simulation (whether it is transfered directly to the file system or goes through dedicated cores) divided by the amount of time it takes for this data to be stored.

Figure 11 presents the aggregate throughput obtained by CM1 with the three approaches on Kraken. At the largest scale (9216 cores) Damaris achieves an aggregate throughput about 6 times higher than the file-per-process approach, and 15 times higher than collective I/O. The results obtained on 672 cores of Grid'5000 are presented in Table 2. The throughput achieved with Damaris here is more than 6 times higher than the other two approaches. Since compression was enabled on BluePrint, we do not provide the resulting throughputs, as it depends on the overhead of the compression algorithm used and the resulting size of the data.

A higher aggregate throughput for the same amount of data represents a shorter utilization time of the network and file system. It reduces the probability that the simulation interfere with other applications concurrently accessing these shared resources, in addition to potentially reducing their energy consumption.

With Nek5000 on the *stremi* cluster of Grid'5000, Table 3 shows that Damaris

	Aggregate throughput
File-per-process	73.5 MB/s
Damaris	337.6 MB/s

Table 3: Average aggregate throughput on Grid'5000's *stremi* cluster, with Nek5000 running on 720 cores.

enables a 4.6× increase of throughput, going from 73.5 MB/s with the file-per-process approach, to 337.6 MB/s with one dedicated core per node.

Conclusion. By avoiding process synchronization and access contention at the level of a node and by gathering data into bigger files, Damaris reduces the I/O overhead, effectively hides the I/O variability and substantially increases the aggregate throughput, thus making a more efficient use of the file system.

4.1.6. Improvements: Leveraging the Spare Time

Section 4.1.4 showed that, with both applications, dedicated cores remain idle most of the time. In order to leverage the spare time in dedicated cores, we implemented two improvements: compression, and transfer delays. These improvements are evaluated hereafter in the context of CM1. Again here, Damaris aggregates data to write one file per dedicated core.

Compression. We used dedicated cores to compress the output data prior to writing it. Using lossless gzip compression, we observed a compression ratio of 1.87 : 1. When writing data for offline visualization, atmospheric scientists can afford to reduce the floating point precision to 16 bits, as it does not visually impact the resulting images. Doing so leads to nearly 6 : 1 compression ratio when coupling with gzip. On Kraken, the time required by dedicated cores to compress and write data was twice longer than the time required to simply write uncompressed data. Yet contrary to enabling compression in the file-per-process approach, the overhead and jitter induced by the compression phase is completely hidden within the dedicated cores, and do not impact the running simulation. In other words, *compression is offered for free* by Damaris.

Data Transfer Delays. Additionally, we implemented in Damaris the capability to delay data movements. The algorithm is very simple and does not involve any communication between processes: each dedicated core computes an estimation of the duration of an iteration of the simulation by measuring the time between two consecutive calls to `damaris_end_iteration` (about 230 seconds on Kraken). This time is then divided into as many slots as there are dedicated cores. Each dedicated core waits for its slot before writing. This avoids access contention at the level of the file system. We evaluated this strategy on 2304 cores on Kraken, the aggregate throughput reaches 13.1 GB/s on average, instead of 9.7 GB/s when this algorithm is not used, which improves the file system utilization and makes dedicated cores spare more time that can be leveraged for other in situ processing tasks.

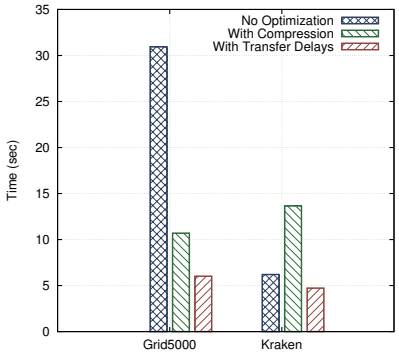


Figure 12: Write time in the dedicated cores when enabling compression or transfer delays.

Summary. These two improvements have also been evaluated on 912 cores of Grid'5000. All results are synthesized in Figure 12, which shows the average write time in dedicated cores. The delay strategy reduces the write time in both platforms. Compression however introduces an overhead on Kraken, thus we are facing a tradeoff between reducing the storage space used or reducing the spare time. A potential optimization would be to enable or disable compression at run time depending on the need to reduce write time or storage space.

4.2. Using Damaris for In Situ Visualization

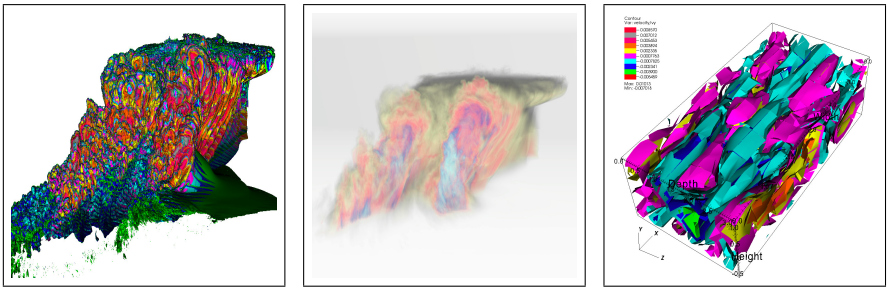
Far from being restricted to performing I/O, Damaris can also leverage the high-level description of data provided in its configuration file to feed in situ visualization pipelines. In the following we evaluate such use of Damaris for in situ visualization. We highlight two aspects: scalability of the visualization algorithms when using dedicated cores, and impact of in situ visualization on application run time.

4.2.1. Platforms and Configurations

We use again the CM1 and Nek5000 applications presented in the previous sections, respectively on Blue Waters and Grid'5000. The platforms and configurations of the experiments are described hereafter.

Blue Waters Blue Waters [50] is a 13.3-petaflops supercomputer deployed at NCSA. It features 26,864 nodes in 237 Cray XE6 cabinets and 44 Cray XK7 cabinets, running Cray Linux Environment (CLE). We leveraged the XE6 nodes, each of which features 16 cores.

Methodology with CM1 on Blue Waters. CM1 requires a long run time before an interesting atmospheric phenomenon appears, and such a phenomenon may not appear at small scale. Yet contrary to the evaluation of I/O performance, we need visualizable phenomena to appear in order to evaluate the performance of in situ visualization tasks. Thus we first ran CM1 with the help of atmospheric scientists to produce relevant data. We generated a representative dataset of $3840 \times 3840 \times 400$ points spanning several iterations.



(a) CM1 Isosurface (b) CM1 Ray Casting (c) Nek5000 Isosurface

Figure 13: Example results obtained in situ with Damaris: (a) 10-level isosurface of the DBZ variable on 6400 cores (Blue Waters). (b) Ray-casting of the *dbz* variable on 6400 cores (Blue Waters). (c) Ten-level isosurface of the y velocity field in the TurbChannel configuration of Nek5000.

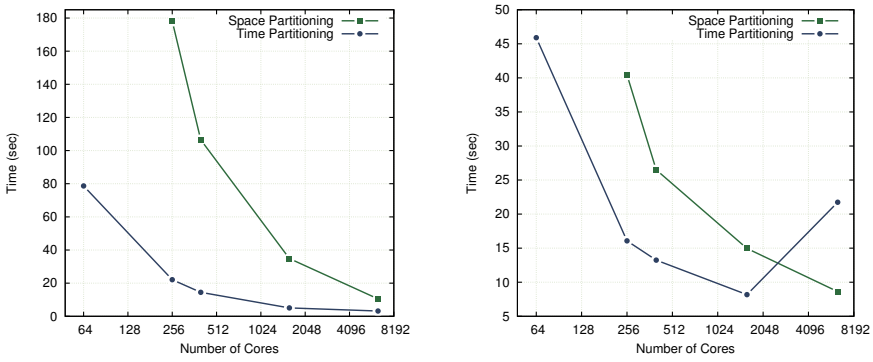
We then extracted the I/O kernel from the CM1 code and built a program that replays its behavior at a given scale and with a given resolution by reloading, redistributing and interpolating the precomputed data. The I/O kernel, identical to the I/O part of the simulation, calls Damaris functions to transfer the data to Damaris. Damaris then performs in situ visualization through a connection to VisIt's *libsim* library [30], either in a time-partitioning manner or using dedicated cores. Our goal with CM1 is to show the interplay between the scalability of the visualization tasks and the use of dedicated cores to run them.

Methodology with Nek5000 on Grid'5000. With Nek5000, we used the *stremi* cluster of Grid'5000 already presented in the previous section. In addition to the *MATiS* configuration, we also use the *turbChannel* configuration, which runs at smaller scales and is more appropriate for interactive in situ visualization. Our goal with Nek5000 is to show the impact of in situ visualization on the variability of the application's run time.

Using Damaris in Time-Partitioning Mode. In order to compare the traditional "time-partitioning" approach with the use of dedicated cores enabled by Damaris, we added a time-partitioning mode in Damaris. This mode, which can be enabled through the configuration file, prevents Damaris from dedicating cores, and runs all plugins in a synchronous manner on all cores running the simulation. This mode allows us to compare the traditional time-partitioning in situ visualization approach with the use of dedicated cores without having to modify the simulations twice.

4.2.2. Impact of Dedicated Cores on the Scalability of Visualization Tasks

With CM1 on Blue Waters, we measured the time (average of 15 iterations) to complete either an isosurface rendering or a ray casting rendering using time partitioning and dedicated cores for each scenario. The comparative results are reported in Figure 14.



(a) In Situ Isosurface

(b) In Situ Ray Casting

Figure 14: Rendering time using ray-casting and isosurfaces, with time-partitioning and dedicated cores with CM1. Note that the number of cores represents the total number used for the experiments; using a dedicated-core approach, 1/16 of this total number is effectively used for in situ visualization, which explains the overall higher rendering time with dedicated cores.

The isosurface algorithm (resulting image presented in Figure 13 (a)) scales well with the number of cores using both approaches. A time-partitioning approach would thus be appropriate if the user does not need to hide the run time impact of in situ visualization. However, on 6400 cores, it takes as much time to complete the rendering as on 400 dedicated cores. In terms of pure computational efficiency, an approach based on dedicated cores is thus 16 times more efficient.

The ray-casting algorithm (resulting image presented in Figure 13 (b)) on the other hand has a poorer scalability. After decreasing, the rendering time goes up again at a 6400-core scale, and it becomes about twice more efficient to use a reduced number of dedicated cores to complete this same rendering.

Conclusion. The choice of using dedicated cores versus a time-partitioning in situ visualization approach depends on (1) the intended visualization scenario, (2) the scale of the experiments and (3) the intended frequency of visual output. Our experiments show that at small scale, the performance of rendering algorithms are good enough to be executed in a time-partitioning manner, provided that the user is ready to increase the run time of his simulation. At large scale however, it becomes more efficient to use dedicated cores, especially when using ray-casting, where the observed rendering performance is substantially better when using a reduced number of processes.

4.2.3. Impact of In Situ Visualization on Run Time Variability

Our goal in this series of experiments is to show the impact of in situ visualization tasks on the run-time variability of the simulation, and to show how dedicated cores help alleviate this variability. We show in particular the effect of interactivity on this variability. We use Nek5000 for this purpose.

Figure 13 (c) shows the result of a 10-level isosurface rendering of the fluid velocity along the y axis, with the TurbChannel case. We use the MATiS configuration to show the scalability of our approach based on Damaris against a standard, time-partitioning approach.

Results with the TurbChannel Configuration. To assess the impact of in situ visualization on the run time, we run TurbChannel on 48 cores using the two approaches: first we use a time-partitioning mode, in which all 48 cores are used by the simulation and synchronously perform in situ visualization. Then we switch on one dedicated core per node, leading to 46 cores being used by the simulation while 2 cores asynchronously run the in situ visualization tasks.

In each case, we consider four scenarios:

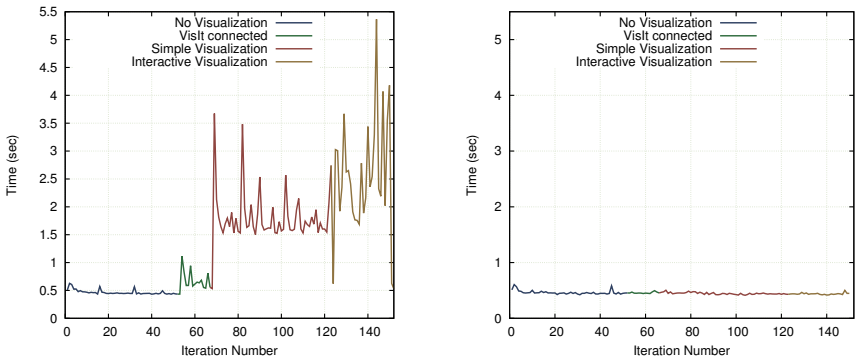
1. The simulation runs without visualization;
2. A user connects VisIt to the simulation but does not ask for any output;
3. The user asks for isosurfaces of the velocity fields but does not interact with VisIt any further (letting the Damaris/Viz update the output after each iteration);
4. The user has heavy interactions with the simulations (for example rendering different variables, using different algorithms, zooming on particular domains, changing the resolution).

Figure 15 presents a trace of the duration of each iteration during the four aforementioned scenarios using the two approaches. Figure 15 (a) shows that in situ visualization using a time-partitioning approach has a large impact on the simulation run time, even when no interaction is performed. The simple act of connecting VisIt without rendering anything forces the simulation to at least update metadata at each iteration, which takes time. Figure 15 (b) shows that in situ visualization based on dedicated cores, on the other hand, is completely transparent from the point of view of the simulation.

Results with the MATiS Configuration. We ran the MATiS configuration on 816 cores of the *stremi* cluster. Each iteration takes approximately one minute and due to the size of the mesh, it is difficult to perform interactive visualization. Therefore we connect VisIt and simply query for a 3D pseudo-color plot of the vx variable (x component of the fluid velocity) that is then updated at desired iterations.

For the following results, the time-partitioning approach initially outputs one image every time step, while dedicated cores adapted the output frequency to one image every 25 time steps in order to avoid blocking the simulation when the shared memory buffer becomes full. To conduct a fair comparison, we thus setup the time-partitioning mode such that it outputs one image every 25 iterations.

Figure 16 reports the behavior of the application with and without visualization performed, and with and without dedicated cores, for the configurations described above. Corresponding statistics are presented in Table 4.



(a) Time-Partitioning (b) Dedicated Cores

Figure 15: Variability in run time induced by different scenarios of in situ interactive visualization.

Table 4: Average iteration time of the Nek5000 MATiS configuration with a time-partitioning approach and with dedicated cores, with and without visualization.

Iteration Time		Average	Std. dev.
Time Partitioning	w/o vis.	75.07 sec	22,93
	with vis.	83.16 sec	43.67
Space Partitioning	w/o vis.	67.76 sec	20.09
	with vis.	64.79 sec	20.44

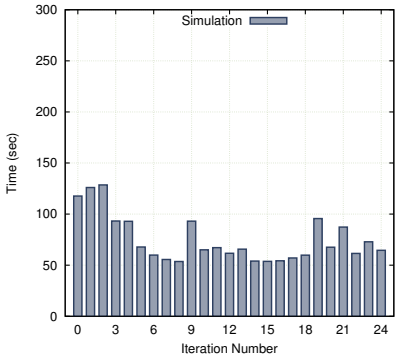
Conclusion. Time-partitioning visualization not only increases the average run time but also increases the standard deviation of this run time, making it more unpredictable. On the other hand, the approach based on dedicated cores yields more consistent results. One might expect dedicated cores to interfere with the simulation as it performs intensive communications while the simulation runs. However, in practice we observe very little such run time variation.

We also remark that decreasing the number of cores used by the simulation can actually decreases its run time. Nek5000 on Grid'5000, for instance, has to run with a number of nodes that is too large, in order to have enough memory.

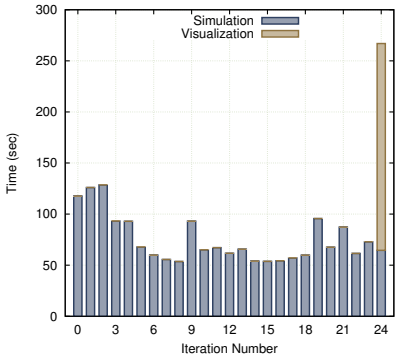
5. Discussion: Dedicated Cores vs. Dedicated Nodes

Two important questions can be asked about approaches like Damaris, which propose to dedicate cores for data processing and I/O.

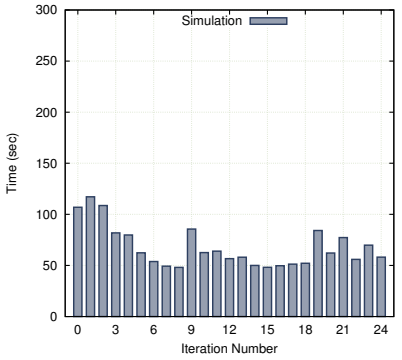
- How many dedicated cores should be used?
- How does dedicating cores compares with dedicating nodes?



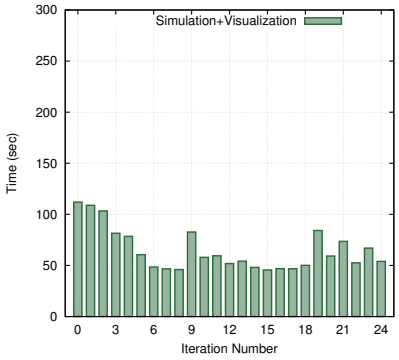
(a) Time Partitioning, w/o Visualization



(b) Time Partitioning, with Visualization performed every 25 time step



(c) Space Partitioning, w/o Visualization



(d) Space Partitioning, with Visualization performed every 25 time steps

Figure 16: Iteration time of the MATiS configuration with a time partitioning approach (top) or a space partitioning approach (bottom), without visualization (left), with visualization (right).

In this section we propose to answer these two questions through experiments with the CM1 and Nek5000 simulations on Grid'5000. We implemented in Damaris the option to use dedicated nodes instead of dedicated cores. Some details of this implementation are given hereafter, before diving into our experimental results.

We restrict our study to I/O. The choice of dedicating cores over dedicating nodes for in situ visualization indeed depends on too many parameters (including the amount of data involved, the simulation, the platform, and most importantly the visualization scenarios) and deserves an entire study that we reserve for a future work.

5.1. Dedicated Nodes in Damaris

In order to compare dedicated cores with dedicated nodes, we either needed a state-of-the-art framework that provides dedicated nodes, such as DataSpace [55], or to implement dedicated nodes inside the Damaris framework. We chose the later because (1) our simulations are already instrumented with Damaris' API, allowing us to switch between each approach without having to modify the simulation with another framework's API, and (2) comparing the use of dedicated cores in Damaris with the use of dedicated nodes in another framework would make it harder to distinguish performance benefits coming from the approach (dedicated cores vs. dedicated nodes) from performance benefits coming from specific optimizations of the framework itself. This following section gives an overview of our implementation of dedicated nodes in Damaris.

5.1.1. Implementation

The implementation of dedicated nodes in Damaris relies on asynchronous MPI communications through Damaris' Distributed Reactor. Each simulation core is associated with a server running in a dedicated node. A dedicated node hosts one server on each of its cores. Different simulation cores may thus interact with the same dedicated node, but with a different core (a different server) in this node.

When a client calls `damaris_write`, it first sends an event to its associated server. This event triggers a `RemoteWrite` callback in the server. When the server enters this callback, it starts a blocking `receive` to get the data sent by the client. The client sends its data to the server, along with metadata information such as the *id* of the variable to which the data belongs. A buffer is maintained in clients to allow these transfers to be non-blocking. When the client needs to send data to dedicated nodes, it copies the data into this buffer and issues a non-blocking `send` to the server using the copied data (note that this communication phase is non-blocking in clients, but blocking on servers). The status of this operation is checked in later calls to the Damaris API and the buffer is freed when the transfer is completed.

Other solutions exist in the literature, for example using RDMA [56] (remote direct memory access). We chose to use simple asynchronous communications for simplicity and portability. The flexibility of our design, along with the recent addition of dynamic RDMA windows in the MPI 3 standard, would ease such an RDMA-based implementation in Damaris in a near future.

5.1.2. "Switching Gears"

Switching between dedicated cores and dedicated nodes, as well as changing the number of dedicated resources, can be done through the configuration, without recompiling the application.

- `<dedicated cores="n" nodes="0"/>` enables n dedicated cores per node. In our current implementation of Damaris, the number of cores per node must divide evenly into the number of dedicated cores.

- `<dedicated cores="0" nodes="n"/>` enables n dedicated nodes. The total number of nodes must divide evenly into the number of dedicated nodes.
- `<dedicated cores="0" nodes="0"/>` disables dedicated cores and nodes. It triggers the time-partitioning mode.

This configuration would allow for a hybrid approach that uses both dedicated cores and dedicated nodes. However this approach is not supported by Damaris yet, as we haven't found any real-life scenario that would benefit from it.

The implementation of all three approaches –time partitioning, dedicated cores, dedicated nodes– within the same framework allows us to evaluate their respective performance in the next sections.

5.2. Dedicated Core(s) vs. Dedicated Nodes: an Experimental Insight

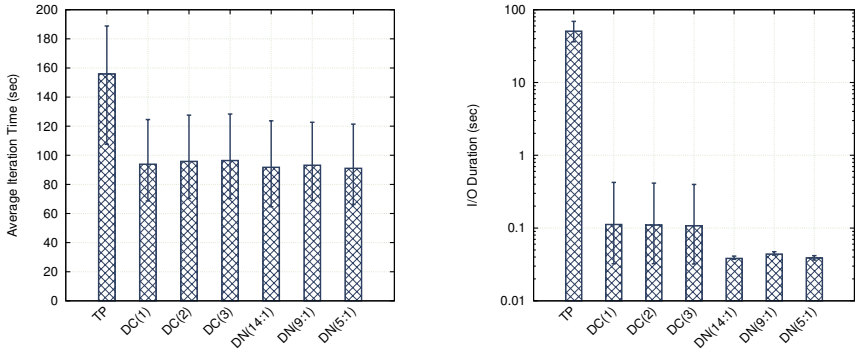
In the following, we present the results obtained with the Nek5000 and CM1 simulations, using the different modes in which Damaris can now operate.

5.2.1. Results with the Nek5000 Application

We used the MATiS configuration of Nek5000 and ran it on 30 nodes (720 cores) of the Grid'5000's *stremi* cluster. We deploy PVFS on 4 additional nodes of this cluster. All nodes (including the file system) communicate through a 1G Ethernet network.

Nek5000 initially wrote most of its checkpoint/restart data in the form of ASCII files, which appeared to be highly inefficient compared to using a high-level data format such as HDF5. We thus rewrote its I/O part as an HDF5-based plugin for Damaris, and used Damaris in 7 configurations: without dedicated resources (time partitioning, abbreviated TP), using 1, 2, or 3 dedicated cores per node (abbreviated DC(1), DC(2) and DC(3)), and using 2, 3 or 5 dedicated nodes (DN(14:1), DN(9:1), DN(5:1) respectively, where the notation $x : y$ represents the ratio of computation nodes to dedicated nodes). Despite the different number of simulation cores in each configuration, the same mesh is used as input for Nek5000 and, therefore, the same amount of data is produced (about 3.5 GB per iteration). We run Nek5000 for 10 such iterations in each configuration.

Overall run time. All configurations based on dedicated resources enable a 40% decrease of overall run time compared with the time-partitioning configuration. Note that because of the inherent variability of the duration of the computation phases within a single iteration (represented in Figure 17 (a) by the minimum and maximum iteration times), it is not possible to tell which of the configuration is actually the best one. Considering these results only, we can argue that using more dedicated cores or more dedicated nodes is potentially an advantageous choice (as long as the efficiency of running your simulation is not affected) because it offers more resources for post-processing and I/O tasks. The choice



(a) Average iteration time

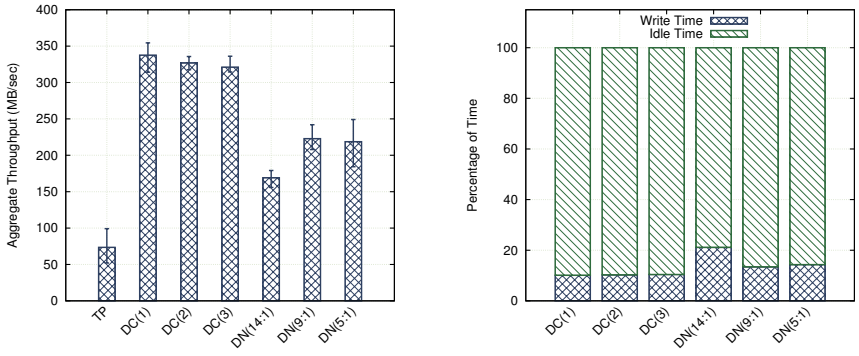
(b) Average I/O time

Figure 17: Experiment with Nek5000 on 720 cores of Grid'5000 *stremi* cluster. Damaris is configured to use either no dedicated resources (TP), x =1, 2 or 3 dedicated cores (DC(x)), or a ratio of x computation nodes to y dedicated nodes (DN($x : y$)). We report (a) the average, maximum and minimum time of a single iteration (computation+I/O), and (b) the average, maximum and minimum time (logarithmic scale) of an I/O phase from the point of view of the simulation.

of using dedicated cores or dedicated nodes can then be based on the characteristics of these post-processing time (scalability, memory requirement, execution time, etc.).

I/O impact. Figure 17 (b) shows that the duration of the I/O phase as perceived by the simulation becomes negligible when using an approach based on dedicated resources. Dedicated cores reduce this time down to about 0.1 seconds, while dedicated nodes reduce it to about 0.04 seconds. This difference in communication time between dedicated cores and dedicated nodes can be easily explained. When using dedicated cores, the client competes with other clients for the access to a mutex-protected segment of shared memory. When using dedicated nodes on the other hand, this contention does not occur, as each client simply makes a local copy of its data and issues a non-blocking send that proceeds in parallel with the simulation. Therefore, while the I/O phase appears faster with dedicated nodes, our results do not show the potential impact that background communications with dedicated nodes may have on the performance of the simulation.

Aggregate throughput. From the point of view of writer processes (or from the point of view of the file system), the different configurations lead to different aggregate throughput. Figure 18 (a) shows that dedicated cores achieve the highest throughput. This throughput is slightly degraded as the number of dedicated cores per node increases, due to contention between dedicated cores on the same node. Dedicated nodes also increase the aggregate throughput compared with time partitioning, but do not achieve the throughput of dedicated



(a) Average aggregate throughput

(b) Spare time in dedicated resources

Figure 18: Experiment with Nek5000 on 720 cores of Grid'5000 *stremi* cluster. Damaris is configured to use either no dedicated resources (TP), x =1, 2 or 3 dedicated cores (DC(x)), or a ratio of x computation nodes to y dedicated nodes (DN($x : y$)). We report (a) the average, maximum and minimum aggregate throughput from writer processes, and (b) the spare time in dedicated processes for the approaches that leverage them.

cores. This is due to the fact that all cores in dedicated nodes are writing, and thus compete for the network access at the level of each single dedicated nodes. Additionally, the lower throughput observed when using only two dedicated nodes can be explained by the fact that the file system features four data servers. Therefore, dedicating only two nodes does not fully take advantage of parallelism across writers.

Spare time. Finally, Figure 18 (b) shows the spare time in dedicated resources. In all configurations based on dedicated cores, the dedicated cores spend 10% of their time writing, and remain idle 90% of the time. Dedicated nodes spend slightly more time writing (from 13 to 20% of their time). This is a direct consequence of the difference in aggregate throughput.

Conclusion. Overall, all the configurations based on dedicated resources improve the simulation run time in a similar way. These configurations however differ in other aspects. By avoiding contention at the level of a node, dedicated cores achieve a higher throughput and therefore spare more time that can be used for data processing. Yet, if we weight this spare time by the number of cores that can be used to leverage it (90 when dedicating 3 cores per node, 120 when dedicating 5 nodes), the configuration based on 5 dedicated nodes appears to spare more resources (core-seconds) in spite of sparing less time per core.

The choice of whether one should use an approach based on dedicated cores or dedicated nodes is of course not restricted to these considerations. Some memory-bound simulations may not be able to afford allocating shared memory to dedicated cores, and would rather benefit from dedicated nodes. Some I/O in-

tensive simulations on the other hand may not be able to transfer large amounts of data to a reduced number of dedicated nodes and will prefer dedicated cores.

5.2.2. Results with the CM1 application

In this section, we leverage experiments with the CM1 simulation to show that the choice of one approach over another also depends on the platform considered.

We used CM1 on Grid'5000's Nancy and Rennes sites. On the Nancy site we use the *graphene* cluster. Each node of this cluster consists of a 4-core Intel Xeon 2.53 GHz CPU with 16 GB of RAM. Intra-cluster communication is done through a 1G Ethernet network. A 20G InfiniBand network is used between these nodes and the OrangeFS file system deployed on 6 I/O servers.

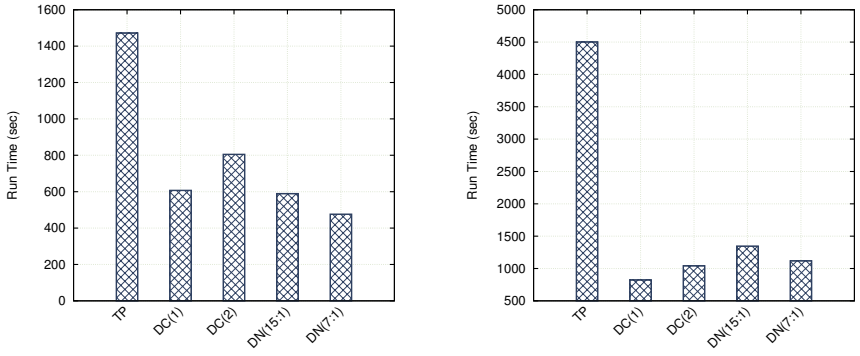
On the Rennes site we use the *parapluie* cluster, already presented in Section 4.1. The nodes communicate with one another through a 1G Ethernet network and with an OrangeFS file system deployed on 3 servers across a 20G InfiniBand network.

We deploy CM1 on 32 nodes (128 cores) on the Nancy site. On the Rennes site, we deploy it on 16 nodes (384 cores). In both cases, we configure CM1 to complete 2520 time steps. We vary its output frequency, using 10, 20 or 30 time steps between each output. Damaris is configured to run with CM1 in five different scenarios that cover the three I/O approaches considered: time partitioning, dedicated cores (one or two – DC(1) and DC(2)), and dedicated nodes using a ratio of 7:1 (DN(7:1), 7 compute nodes for one dedicated node) or 15:1 (DN(15:1), 15 compute nodes for one dedicated node). DN(7:1) thus uses four dedicated nodes on the Nancy site, two on the Rennes site. DN(15:1) dedicates two nodes on the Nancy site, one on the Rennes site.

Impact of the platform. Figure 19 shows that in both clusters, dedicating resources drastically improve the performance of CM1 compared with a time-partitioning approach. Dedicating four nodes on Nancy enables an almost 3× overall speedup, while dedicating one core in each node on the Rennes cluster leads to more than 5× speedup. Our results also show that the best approach in terms of overall run time depends on the platform. It consists of using dedicated nodes with a 7:1 ratio on the Nancy cluster, and using one dedicated core per node on the Rennes cluster. This conclusion is not surprising, since the Nancy cluster only provides 4 cores per node. Dedicating some of these cores thus has a large impact on the simulation. On the Rennes cluster, which provides 24 cores per node, dedicating some of these cores does not remove such an important fraction of computational power from the simulation and is thus more efficient than dedicating nodes.

5.3. Conclusion

Over the years, several research groups have proposed new approaches to I/O and data processing based on dedicated resources. These approaches can be divided into a group of approaches based on dedicated cores, and a group of approaches based on dedicated nodes. While Damaris was initially part of



(a) Run time of CM1 on Nancy cluster

(b) Run time of CM1 on Rennes cluster

Figure 19: Experiment with CM1 on Grid'5000 Rennes (24 cores per node) and Nancy (4 cores per node) sites. Damaris is configured to use either no dedicated resources (TP), x =1 or 2 dedicated cores (DC(x)), or a ratio of 7:1 or 15:1 dedicated nodes (DN(7:1) and DN(15:1)). We report total run time for 2520 time steps.

the first one, we extended it to support a wider range of configurations. It now offers to dedicate either a subset of cores in each multicore node, or entire nodes. Additionally, it also offers to not dedicate any resource at all, performing all data processing and movement synchronously. This flexibility, made possible in particular through a configuration file that allows us to switch between modes very easily, let us to compare these approaches.

Our results show that dedicating resources for I/O is a highly efficient method to improve the I/O performance of a simulation, both in terms of overall run time, aggregate throughput and performance variability. They also highlighted the fact that there is no clear advantage of one approach over the other: dedicating cores appears more efficient than dedicated nodes under certain conditions, and the opposite holds under different conditions. The choice of one approach over the other may also depend on criteria other than the overall run time. Our experiments with Nek5000 showed that while this run time is very similar under the different approaches, the resulting aggregate throughput favors dedicating cores, while the resulting spared resources (spare time $\times$ number of cores in dedicated resources) advocates for using dedicated nodes. Our experiments with CM1 showed that the choice of one approach over the other also depends on the platform. While an approach based on dedicated cores is more suitable on a platform featuring a large number of cores per node, it may be more efficient to use dedicated nodes on a platform with a reduced number of cores per node.

6. Related Work

In this section, we position our work with respect to related work. We start by discussing approaches that attempt at improving I/O performance. We then examine approaches to in situ visualization.

6.1. Damaris in the "I/O Landscape"

Through its capability of gathering data into larger buffers and files, Damaris can be compared to the data aggregation feature in ROMIO [53]. This feature is an optimization of Collective I/O that leverages a subset of processes, called "aggregators", to actually perform the I/O on behalf of other processes. Yet, data aggregation is performed synchronously in ROMIO: all processes that do not perform actual writes in the file system must wait for the aggregator processes to complete their operations. Besides, aggregators are not dedicated processes, they run the simulation after completing their I/O. Through dedicated cores, Damaris can perform data aggregation and potential transformations in an asynchronous manner and still use the idle time remaining in the dedicated cores.

Other efforts focus on overlapping computation with I/O in order to reduce the impact of I/O latency on overall performance. Overlap techniques can be implemented directly within simulations [57], using asynchronous communications. Non-blocking I/O primitives started to appear as part of the current MPI 3 standard, these primitives are still implemented as blocking in practice.

Other approaches leverage data-staging and caching mechanisms [24, 58], or forwarding approaches [26] to achieve better I/O performance. Forwarding architectures run on top of dedicated resources in the platform, which are not configurable by the end-user, that is, the user cannot run custom data processing in forwarding resources. Similarly to the parallel file system, these dedicated resources are shared by all users. This leads to cross-application access contention and thus, to I/O variability. However, the trend toward I/O delegate systems underlines the need for new I/O approaches. Our approach relies on dedicated I/O cores at the application level, or dedicated nodes bound to the application, rather than relying on hardware I/O-dedicated or forwarding nodes, with the advantage of letting users configure their dedicated resources to best fit their needs.

The use of local memory to alleviate the load on the file system is not new. The Scalable Checkpoint/Restart (SRC) by Moody et al. [59] already makes use of node-level storage to avoid the heavy load caused by periodic global checkpoints. Yet their work does not use dedicated resources or threads to handle or process data, and the checkpoints are not asynchronous.

Dedicated-Core-Based Approaches. Closest to our work are the approaches by Li et al. [22], and Ma et al. [29]. While the general goals of these approaches are similar (leveraging service-dedicated cores for non-computational tasks), their design is different, and so is the focus and the (much lower) scale of their evaluation. The first one mainly explores the idea of using dedicated cores in conjunction with SSDs to improve the overall I/O throughput. Architecturally, it relies

on a FUSE interface, which introduces unnecessary copies through the kernel and reduces the degree of coupling between cores. Using small benchmarks we noticed that such a FUSE interface is about 10 times slower in transferring data between cores than using shared memory. In the second, active buffers are handled by dedicated processes that can run on any node and interact with cores running the simulation through the network. In contrast to both approaches, Damaris makes a much more efficient design choice using the shared intra-node memory, thereby avoiding costly copies and buffering. The approach defended by Li et al. is demonstrated on a small 32-node cluster (160 cores), where the maximum scale used in the work by Ma et al. is 512 cores on a Power3 machine, for which the overall improvement achieved for the global run time is marginal. Our experimental analysis is much more extensive and more relevant for today's scales of HPC simulations: we demonstrated the excellent scalability of Damaris on a real supercomputer (Kraken, ranked 11^{th} in the Top500 supercomputer list at the time of the experiments) with up to almost 10,000 cores, and with the CM1 tornado simulation, one of the target applications of the Blue Waters post-Petascale supercomputer project. We demonstrated not only a speedup in I/O throughput by a factor of 15 (never achieved by previous approaches), but we also showed that Damaris totally hides the I/O jitter and substantially cuts down the application run time at such high scales. With Damaris, the execution time for CM1 at this scale is even divided by 3.5 compared to approaches based on collective I/O! Moreover, we further explored how to leverage the spare time of the dedicated cores. We demonstrated for example that it can be used to compress data by a factor of 6.

6.2. Damaris in the "In Situ Visualization Landscape"

Loosely-Coupled Visualization Strategies. Ellsworth et al. [60] propose to use distributed shared memory (DSM) to avoid writing files when performing concurrent visualization. Such an approach has the advantage of decoupling the simulation and visualization processes, but reading data from the memory of the simulation's processors can increase run time variability. The scalability of a distributed shared memory design is also a limiting factor.

Rivi et al. [61] introduce the ICARUS plugin for ParaView together with a description of VisIt and ParaView's in situ visualization interfaces. ICARUS employs an HDF5 DSM file driver to ship data to a distributed shared memory buffer that is used as input to a ParaView pipeline. This DSM stores a view of the HDF5 files that can be concurrently accessed by the simulation and visualization tools. The HDF5 API allows to bridge the simulation and ParaView with minimum code changes (provided that the simulation already uses HDF5), but it produces multiple copies of the data and a complete transformation of data into an intermediate HDF5 representation. Also, the visualization library on the remote resource requires the original data to conform to this HDF5 representation. Damaris, on the other hand, is not based on any data format and efficiently leverages shared-memory to avoid as much as possible unnecessary copies of data. Besides, its API is simpler than that of HDF5 for simulations that do not already use HDF5.

Malakar et al. [35] present an adaptive framework for loosely-coupled visualization, in which data is sent over a network to a remote visualization cluster at a frequency that is dynamically adapted depending on resource availability. Our approach also adapts output frequency to resource usage.

The PreDatA [33] middleware proposes to dedicate a set of nodes as a staging area to perform a first step of data processing prior to I/O for the purpose of subsequent visualization. The coupling between the simulation and the staging area is done through the ADIOS [43] I/O layer. The use of the ADIOS backend allows to decouple the simulation and the visualization by simply integrating data analysis as part of an existing I/O stack [62]. While Damaris borrows the use of an XML file from ADIOS in order to simplify its API, it makes the orthogonal choice of using dedicated cores rather than dedicated nodes. Thus it avoids potentially costly data movements across nodes.

GLEAN [34] provides in situ visualization capabilities with dedicated nodes. The authors use the PHASTA simulation on the Intrepid supercomputer and ParaView for analysis and visualization on the Eureka machine. Part of the analysis in GLEAN is done in a time-partitioning manner at the simulation side, which makes it a hybrid approach involving tightly- and loosely-coupled in situ analysis. Our approach shares some of the same goals, namely to couple a simulation with run-time visualization, but we run the visualization tool on one core of the same node instead of dedicated nodes. GLEAN is also used in conjunction with ADIOS [63].

EPSN [44] is an environment providing steering and visualization capabilities to existing parallel simulations. Simulations instrumented with EPSN ship their data to a visualization pipeline running on a remote cluster, thus EPSN is an hybrid approach including both code changes and the use of additional remote resources. In contrast to EPSN, all visualization tasks using Damaris can be performed on dedicated cores, closer to the simulation, thus reducing the network overhead.

Zheng et al. [64] have provided a model to evaluate the tradeoff between in situ synchronous visualization and loosely-coupled visualization through staging areas. This model can be applied to compare in situ using dedicated cores instead of remote resources, with the difference being that approaches utilizing dedicated cores do not have network communication overhead.

Tightly-Coupled In Situ Visualization. When it comes to tightly integrate analysis tasks in simulations codes, the existing solutions often do not meet all of the requirements presented in Section 2.

SciRun [65] is a complete computational-steering environment that includes visualization. Its in situ capabilities can be used with any simulation implemented with SciRun solvers and structures. SciRun is an example of the trend towards integrating visualization, data analysis and computational steering in the simulation process. Simulations are written specifically for use in SciRun in order to exchange data with zero data copy, but adapting an existing application to this framework can be a daunting task.

DIY [66] offers a number of communication primitives allowing to easily build

efficient parallel in situ analysis and visualization algorithms. However it does not aim to provide a way to dedicate resources on which to run these algorithms. DIY could therefor very well be coupled with Damaris to implement powerful in situ analysis algorithms while Damaris provides the flexibility of running them on dedicated resources.

Tu et al. [67] propose an end-to-end approach for an earthquake simulation using the Hercule framework. All the components of the simulation, including visualization, run in parallel on the same machine, and the only output consists of a set of JPEG files. The data processing tasks in Hercule are still performed in a synchronous manner, and any operation initiated by a process to perform these tasks impacts the performance of the simulation.

In the context of ADIOS, CoDS (Co-located DataSpaces) [68] builds a distributed object-based data space abstraction and can use dedicated nodes (and recently dedicated cores with shared memory) with PreDatA, DataStager and DataSpace. ADIOS+CoDS has also been used for code coupling [69] and demonstrated with different simulation models. While the use of dedicated cores to accomplish two different tasks is a common theme in our approach, our objective in this chapter was to compare the performance impact on the simulation of a collocated visualization task with a directly embedded visualization. Besides, placement of data in shared memory in the aforementioned works is done through the ADIOS interface, which creates a copy of data from the simulation to the shared memory using a file-writing interface. We leverage the double-buffering technique usually implemented in simulations as an efficient alternative for sharing data.

Posteriorly to our work, Dreher and Rafin [70] built on the FlowVR framework (initially proposed for real-time interactive parallel visualization in the context of virtual reality) to provide a solution integrating both time partitioning, dedicated cores and dedicated nodes. They address usability by providing a simple *put*/*get* interface and a Python script that describes the various component of the visualization pipeline. They went one step further by providing in situ interactive simulation steering in a cave-like system with haptic devices [71], highlighting a case where the simulation process and research are part of the same workflow.

7. Conclusion and Future Directions

As HPC resources exceeding millions of cores become a reality, science and engineering codes invariably must be modified in order to efficiently exploit these resources. An important challenge in maintaining high performance is data management, which nowadays does not only include writing and storing data efficiently, but also analyzing and visualizing these data in order to retrieve a scientific insight.

This paper provides a comprehensive overview of Damaris, an approach which proposes to offload data management tasks, including I/O, post-processing and visualization, into dedicated cores of multicore nodes. Damaris efficiently leverages shared-memory to improve memory usage when transferring data from

cores running the simulation to cores running data-related tasks. Thanks to its plugin system and an external description of data, Damaris is highly adaptable to a wide range of simulations.

We first used Damaris to offload I/O tasks in dedicated cores, and compared the resulting performance with the two standard approaches to I/O in HPC simulations: the File-per-process and the Collective I/O approaches. By gathering I/O operations in a reduced number of cores and by avoiding synchronization between these cores, Damaris is able to completely hide all I/O-related costs, and in particular the I/O variability. Our experiments using the CM1 atmospheric simulation and the Nek5000 computation fluid dynamic, in particular on up to 9216 cores of the Kraken supercomputer, showed that Damaris can achieve a 15 times higher throughput compared with the collective I/O approach. Damaris also dramatically reduces the application run time, leading to a 3.5× speedup in CM1, for example. Observing that dedicated cores still remain idle a large fraction of the time, we implemented several improvements, including an overhead-free data compression that achieved up to 600% compression ratio.

We then leveraged the time spared by Damaris on dedicated cores by extending it to support in situ visualization through a connection with the VisIt visualization software. We evaluated our Damaris-based in situ visualization framework on the Grid'5000 and Blue Waters platforms. We showed that Damaris can fully hide the performance variability induced by in situ visualization tasks as well, even in scenarios involving interactions with a user. Besides, Damaris reduces visualization-related code modifications to a minimum in existing simulations.

Finally we further extended Damaris to support the use of dedicated nodes instead of dedicated cores. Based on our framework, we performed a thorough comparison of the dedicated cores, dedicated nodes and time-partitioning approaches for I/O on 3 different clusters of the Grid'5000 testbed, with the CM1 and Nek5000 simulations. Our evaluation shows that approaches based on dedicated resources always perform better than the time-partitioning approach for the selected simulations. They both manage to hide the I/O-related costs and, as a result, improve the overall simulation performance. While the choice of an approach based on dedicated cores over an approach based on dedicated nodes is primarily driven by the number of cores per node available in the platform, this choice also depends on the scalability of the application, its memory usage, and the potential use of spare time in dedicated resources.

To our knowledge, Damaris is the first middleware available to the community[6] that offers the use of dedicated cores or dedicated nodes to serve data management tasks ranging from I/O to in situ visualization. This work paves the way for a number of new research directions with high potential impact. Our study of in situ visualization using Damaris and CM1 revealed that in some simulations such as climate models, an important fraction of the data produced by

[6]See `http://damaris.gforge.inria.fr`

the simulation does not actually contain any part of the phenomenon that are of interest to scientists. When visualizing this data in situ, it thus becomes possible to lower the resolution of non-interesting parts in order to increase the performance of the visualization process, an approach that we call "smart in situ visualization". Challenges to implement smart in situ visualization include automatically discriminating relevant and non-relevant data within the simulation while this data is being produced. This detection should be made without user intervention and be fast enough to not diminish the overall performance of the visualization process. The plugin system of Damaris together with its existing connection with the VisIt visualization software provide an excellent ground to implement and evaluate smart in situ visualization.

We also plan to investigate ways to reduce the energy consumption of simulations that use approaches like Damaris. We have already shown that the time spared by dedicated cores in Damaris can be leveraged to compress the data prior to storing it. An immediate question that can be asked is to which extent does compression in Damaris impacts this energy/performance tradeoff. On one hand, compression reduces the amount of data transferred and thus, the network traffic, which leads to lower energy consumption from data movements. On the other hand, compressing data requires more computation time and higher energy consumption as a result of data movement in the local memory hierarchy. Consequently, a promising direction will consist in investigating the tradeoff between energy, performance and compression level.

Acknowledgments

This work was done in the framework of a collaboration between the KerData (Inria Rennes Bretagne Atlantique, ENS Rennes, INSA Rennes, IRISA) team, the National Center for Supercomputing Applications (Urbana-Champaign, USA) and Argonne National Laboratory, within the Joint Inria-UIUC-ANL-BSC-JSC Laboratory for Extreme-Scale Computing (JLESC), formerly Joint Laboratory for Petascale Computing (JLPC). The work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, by the National Center for Atmospheric Research (NCAR) and Central Michigan University. Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). The authors also thank Robert Wilhelmson for his insight on CM1, Dave Semeraro for the discussions regarding in situ visualization using VisIt, Paul Fischer and Aleksandr Obabko for helping understand Nek5000 and providing input datasets.

References

[1] M. Dorier, G. Antoniu, F. Cappello, M. Snir, L. Orf, Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free

I/O, in: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER '12), IEEE, Beijing, China, 2012. URL: http://hal.inria.fr/hal-00715252.

[2] M. Dorier, R. Sisneros, Roberto, T. Peterka, G. Antoniu, B. Semeraro, Dave, Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework, in: Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '13), Atlanta, Georgia, USA, 2013. URL: http://hal.inria.fr/hal-00859603.

[3] NICS, Kraken supercomputer, national institute for computational sciences, http://www.nics.tennessee.edu/computing-resources/kraken, 2015.

[4] G. H. Bryan, J. M. Fritsch, A benchmark simulation for moist nonhydrostatic numerical models, Monthly Weather Review 130 (2002) 2917–2928.

[5] J. W. L. P. F. Fischer, S. G. Kerkemeier, Nek5000 Web page http://nek5000.mcs.anl.gov, 2008.

[6] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, P. Schwan, Lustre: Building a File System for 1000-node Clusters, in: Proceedings of the 2003 Linux Symposium, Citeseer, 2003.

[7] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, A. Koniges, MPI-IO/GPFS an optimized implementation of MPI-IO on top of GPFS, in: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '01), IEEE Computer Society, Los Alamitos, CA, USA, 2001. doi:10.1145/582034.582051.

[8] H. Shan, J. Shalf, Using IOR to analyze the I/O performance for HPC platforms, in: Proceedings of the Cray User Group Conference (CUG '07), Seattle, Washington, USA, 2007.

[9] INRIA, Grid'5000, http://www.grid5000.fr, 2015.

[10] P. H. Carns, W. B. Ligon, III, R. B. Ross, R. Thakur, PVFS: a Parallel File System for Linux Clusters, in: Proceedings of the 4th annual Linux Showcase & Conference - Volume 4, USENIX Association, Berkeley, CA, USA, 2000.

[11] D. Skinner, W. Kramer, Understanding the causes of performance variability in HPC workloads, in: Proceedings of the IEEE Workload Characterization Symposium (IISWC '05), IEEE Computer Society, 2005, pp. 137–149. doi:10.1109/IISWC.2005.1526010.

[12] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, L. Oliker, Parallel I/O performance: From events to ensembles, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '10), 2010. doi:10.1109/IPDPS.2010.5470424.

[13] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, S. Ibrahim, CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '14), Phoenix, Arizona, USA, 2014. URL: http://hal.inria.fr/hal-00916091.

[14] M. Dorier, G. Antoniu, F. Cappello, M. Snir, L. Orf (Joint INRIA/UIUC Laboratory for Petascale Computing, Grid'5000), Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter, Research Report RR-7706, INRIA, 2012. URL: http://hal.inria.fr/inria-00614597.

[15] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, M. Wolf, Managing variability in the IO performance of petascale storage systems, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), IEEE Computer Society, Washington, DC, USA, 2010. doi:10.1109/SC.2010.32.

[16] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, M. Snir, Scheduling the I/O of HPC Applications under Congestion, Rapport de recherche RR-8519, INRIA, 2014. URL: http://hal.inria.fr/hal-00983789.

[17] R. Thakur, W. Gropp, E. Lusk, On implementing MPI-IO portably and with high performance, in: Proceedings of the sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99), ACM, 1999, pp. 23–32.

[18] HDF5, Hierarchical Data Format, http://www.hdfgroup.org/HDF5/, 2015.

[19] M. Folk, A. Cheng, K. Yates, HDF5: A file format and I/O library for high performance computing applications, in: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '99), 1999.

[20] Unidata, NetCDF, http://www.unidata.ucar.edu/software/netcdf/, 2015.

[21] J. Fu, R. Latham, M. Min, C. D. Carothers, I/O threads to reduce checkpoint blocking for an electromagnetics solver on Blue Gene/P and Cray XK6, in: Proceedings of the International workshop on Runtime and Operating Systems for Supercomputers (ROSS '12), 2012.

[22] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, G. Shipman, Functional partitioning to optimize end-to-end performance on many-core architectures, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), IEEE Computer Society, 2010.

[23] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, F. Zheng, DataStager: Scalable data staging services for petascale applications, in: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09), ACM, New York, NY, USA, 2009, pp. 39–48. doi:10.1145/1551609.1551618.

[24] A. Nisar, W. keng Liao, A. Choudhary, Scaling parallel I/O performance through I/O delegate and caching system, in: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08), 2008. doi:10.1109/SC.2008.5214358.

[25] R. Prabhakar, S. Vazhkudai, Y. Kim, A. Butt, M. Li, M. Kandemir, Provisioning a multi-tiered data staging area for extreme-scale machines, in: Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS '11), 2011. doi:10.1109/ICDCS.2011.33.

[26] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, P. Sadayappan, Scalable I/O forwarding framework for high-performance computing systems, in: Proceedings of the IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09., 2009. doi:10.1109/CLUSTR.2009.5289188.

[27] N. T. B. Stone, D. Balog, B. Gill, B. Johanson, J. Marsteller, P. Nowoczynski, D. Porter, R. Reddy, J. R. Scott, D. Simmel, J. Sommerfield, K. Vargo, C. Vizino, PDIO: High-performance remote file I/O for portals enabled compute nodes, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06), 2006. URL: http://www.scientificcommons.org/43489982.

[28] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, C. Maltzahn, On the role of burst buffers in leadership-class storage systems, in: Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST '12), IEEE, 2012.

[29] X. Ma, J. Lee, M. Winslett, High-level buffering for hiding periodic output cost in scientific simulations, IEEE Transactions on Parallel and Distributed Systems (TPDS) 17 (2006) 193–204.

[30] B. Whitlock, J. M. Favre, J. S. Meredith, Parallel in situ coupling of simulation with a fully featured visualization system, in: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '10), Eurographics Association, 2011.

[31] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, K. Jansen, The paraview coprocessing library: A scalable, general purpose in situ visualization library, in: Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '11), 2011.

[32] D. B. Johnston, First-of-a-kind supercomputer at lawrence livermore available for collaborative research, https://www.llnl.gov/news/newsreleases/2014/May/NR-14-05-02.html, 2014.

[33] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, M. Wolf, PreDatA – preparatory data analytics on peta-scale machines, in: Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS '10), 2010. doi:10.1109/IPDPS.2010.5470454.

[34] M. Rasquin, P. Marion, V. Vishwanath, B. Matthews, M. Hereld, K. Jansen, R. Loy, A. Bauer, M. Zhou, O. Sahni, et al., Electronic poster: Co-visualization of full data and in situ data extracts from unstructured grid CFD at 160k cores, in: ACM/IEEE SC Companion, ACM, 2011, pp. 103–104.

[35] P. Malakar, V. Natarajan, S. S. Vadhiyar, An adaptive framework for simulation and online remote visualization of critical climate applications in resource-constrained environments, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), IEEE Computer Society, Washington, DC, USA, 2010. doi:10.1109/SC.2010.10.

[36] M. Hereld, M. E. Papka, V. Vishwanath, Toward simulation-time data analysis and I/O acceleration on leadership-class systems, in: Proceeding of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '11), Providence, RI, 2011.

[37] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, et al., Extreme scaling of production visualization software on diverse architectures, IEEE Computer Graphics and Applications (2010) 22–31.

[38] H. Yu, K. Ma, A study of I/O techniques for parallel visualization, Journal of Parallel Computing 31 (2005).

[39] H. Yu, C. Wang, R. Grout, J. Chen, K.-L. Ma, In situ visualization for large-scale combustion simulations, IEEE Computer Graphics and Applications 30 (2010) 45–57.

[40] K.-L. Ma, C. Wang, H. Yu, A. Tikhonova, In-situ processing and visualization for ultrascale simulations, Journal of Physics: Conference Series 78 (2007).

[41] K.-L. Ma, In situ visualization at extreme scale: Challenges and opportunities, IEEE Computer Graphics and Applications 29 (2009) 14–19.

[42] D. C. Schmidt, Reactor - an object behavioral pattern for demultiplexing and dispatching handles for synchronous events, 1995.

[43] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin, Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS), in: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments, CLADE '08, ACM, New York, NY, USA, 2008. doi:10.1145/1383529.1383533.

[44] A. Esnard, N. Richart, O. Coulaud, A steering environment for online parallel visualization of legacy parallel simulations, in: Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '06), IEEE, 2006, pp. 7–14.

[45] KitWare, eXtensible Data Model and Format (XDMF), http://www.xdmf.org/, 2015.

[46] LLNL, VisIt, Lawrence Livermore National Laboratory, https://wci.llnl.gov/simulation/computer-codes/visit, 2015.

[47] KitWare, ParaView, http://www.paraview.org/, 2015.

[48] ERDC DSRC, Ezviz, http://daac.hpc.mil/software/ezViz/, 2015.

[49] W. Schroeder, L. Avila, W. Hoffman, Visualizing with VTK: a tutorial, IEEE Computer Graphics and Applications 20 (2000) 20–27.

[50] NCSA, Blue waters supercomputer, national center for supercomputing applications, http://www.ncsa.illinois.edu/BlueWaters/, 2015.

[51] C. M. Chilan, M. Yang, A. Cheng, L. Arber, Parallel I/O performance study with hdf5, a scientific data package, TeraGrid 2006: Advancing Scientific Discovery (2006).

[52] ANL, Mpich, http://www.mpich.org, 2015.

[53] R. Thakur, W. Gropp, E. Lusk, Data Sieving and Collective I/O in ROMIO, Symposium on the Frontiers of Massively Parallel Processing (1999) 182.

[54] Top500 list of supercomputers, http://www.top500.org/, 2015.

[55] Rutgers, DataSpace, www.dataspaces.org/, 2015.

[56] C. Docan, M. Parashar, S. Klasky, Enabling high-speed asynchronous data extraction and transfer using DART, Concurrency and Computation: Practice and Experience (2010) 1181–1204.

[57] C. M. Patrick, S. W. Son, M. Kandemir, Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO, Operating Systems Review (SIGOPS) 42 (2008) 43–49.

[58] F. Isaila, J. G. Blas, J. Carretero, R. Latham, R. Ross, Design and evaluation of multiple level data staging for Blue Gene systems, IEEE Transactions on Parallel and Distributed Systems (TPDS) (2010).

[59] A. Moody, G. Bronevetsky, K. Mohror, B. R. de Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), IEEE Computer Society, Los Alamitos, CA, USA, 2010. doi:10.1109/SC.2010.18.

[60] D. Ellsworth, B. Green, C. Henze, P. Moran, T. Sandstrom, Concurrent visualization in a production supercomputing environment, IEEE Transactions on Visualization and Computer Graphics (TVGC) 12 (2006) 997–1004.

[61] M. Rivi, L. Calori, G. Muscianisi, V. Slavnic, In-situ visualization: State-of-the-art and some use cases, PRACE White Paper (2012), http://www.prace-ri.eu/Visualisation (2011).

[62] F. Zheng, J. Cao, J. Dayal, G. Eisenhauer, K. Schwan, M. Wolf, H. Abbasi, S. Klasky, N. Podhorszki, High end scientific codes with computational I/O pipelines: Improving their end-to-end performance, in: Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities (PDAC '11), ACM, New York, NY, USA, 2011, pp. 23–28. URL: http://doi.acm.org/10.1145/2110205.2110210. doi:10.1145/2110205.2110210.

[63] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, et al., Examples of in transit visualization, in: Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities (PDAC '11), ACM, 2011.

[64] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, N. Podhorszki, In-situ I/O processing: A case for location flexibility, in: Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW '11), ACM, New York, NY, USA, 2011, pp. 37–42. URL: http://doi.acm.org/10.1145/2159352.2159362. doi:10.1145/2159352.2159362.

[65] C. Johnson, S. Parker, C. Hansen, G. Kindlmann, Y. Livnat, Interactive simulation and visualization, Computer 32 (1999) 59–65.

[66] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, A. Chaudhuri, Scalable parallel building blocks for custom data analysis, in: Proceedings of Large Data Analysis and Visualization Symposium LDAV'11, Providence, RI, 2011.

[67] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, D. R. O'Hallaron, From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing, in: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '06), ACM, New York, NY, USA, 2006. doi:10.1145/1188455.1188551.

[68] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, M. Parashar, In-situ feature-based objects tracking for large-scale scientific simulations, in: ACM/IEEE SC Companion, IEEE, 2012.

[69] F. Zhang, M. Parashar, C. Docan, S. Klasky, N. Podhorszki, H. Abbasi, Enabling in-situ execution of coupled scientific workflow on multi-core platform, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '12), IEEE, 2012.

[70] M. Dreher, B. Raffin, et al., A flexible framework for asynchronous in situ and in transit analytics for scientific simulations, ACM/IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14) (2014).

[71] M. Dreher, J. Prevoteau-Jonquet, M. Trellet, M. Piuzzi, M. Baaden, B. Raffin, N. Férey, S. Robert, S. Limet, Exaviz: A flexible framework to analyse, steer and interact with molecular dynamics simulations, Faraday Discussions (2014).